A Flexible, Heterogeneous Image Processing Framework for Spaceborne Reconfigurable Data Processing Modules

Von der Fakultät für Elektrotechnik, Informationstechnik, Physik
der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades eines Doktors
der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

von	Dipl.-Ing. Tobias Lange
aus	Hildesheim

eingereicht am:	04.09.2019
mündliche Prüfung am:	28.02.2020
1. Referent:	Prof. Dr.-Ing. Harald Michalik
2. Referent:	Prof. Dr.-Ing. Mladen Berekovic
Druckjahr:	2020

Danksagung

Diese Arbeit entstand überwiegend während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Institut für Datentechnik und Kommunikationsnetze (IDA) in der Arbeitsgruppe „Instrumentenrechner für die Raumfahrt“ an der Technischen Universität Braunschweig. Die Themenstellung ergab sich aus der Mitarbeit an internationalen Forschungsprojekten.
Mein besonderer Dank gilt Herrn Prof. Dr.-Ing. Harald Michalik für die wohlwollende Betreuung meiner Arbeit und die Übernahme des Referats. Herrn Prof. Dr.-Ing. Mladen Berekovic möchte ich für die Übernahme des Koreferats, sowie Herrn Prof. Dr.-Ing. Rolf Ernst für den Vorsitz der Prüfungskommission danken.
Weiterer Dank gilt meinen Institutskollegen für die langjährige Unterstützung, Motivation und die zahlreichen Anregungen und Diskussionen. Dieser Dank gilt auch meinen Kollegen vom Max-Planck Institut für Sonnensystemforschung (MPS) in Göttingen, die stets eine sehr gute Zusammenarbeit ermöglicht haben.

Insbesondere danke ich meiner Familie und meinen Freunden, die mich jederzeit unterstützt und ermutigt haben.

Tobias Lange

Braunschweig, im Mai 2020

Kurzfassung

Wissenschaftliche Instrumente auf aktuellen Raumfahrtmissionen sind oft mit hochauflösenden Sensoren ausgestattet. Insbesondere kamerabasierte Instrumente produzieren dabei eine große Menge an Daten. Diese werden üblicherweise nach dem Empfang auf der Erde weiterverarbeitet, um daraus wissenschaftlich relevante Informationen zu gewinnen. Aufgrund der großen Entfernung von Missionen innerhalb unseres Sonnensystems ist die Datenrate zur Übertragung an die Bodenstation oft sehr begrenzt. Das Volumen der wissenschaftlich relevanten Daten ist meist deutlich kleiner als die aufgenommenen Rohdaten. Daher ist es vorteilhaft, diese bereits an Board der Sonde zu verarbeiten.
Ein Beispiel für solch ein Instrument ist der *Polarimetric and Helioseismic Imager* (*PHI*) an Bord von *Solar Orbiter*. Um die Daten aufzunehmen, zu speichern und zu verarbeiten, ist das Instrument mit einem Data Processing Module (DPM) ausgestattet. Dieses nutzt ein heterogenes Rechnersystem aus einem dedizierten *LEON3* Prozessor, zusammen mit zwei rekonfigurierbaren *Xilinx Virtex-4* Field-Programmable Gate Arrays (FPGAs).
Die folgende Arbeit gibt einen Überblick über verfügbare Komponenten zur Datenverarbeitung (Prozessoren und FPGAs), die den Anforderungen von Raumfahrtmissionen gerecht werden, und stellt einige existierende Plattformen vor, die auf einem heterogenen System aus Prozessor und FPGA basieren. Hierzu gehört auch das Data Processing Module des *PHI* Instrumentes, dessen Architektur im Verlauf dieser Arbeit beschrieben wird. Als Kernelement der Dissertation wird ein Framework vorgestellt, das sowohl eine performante, als auch eine flexible Bilddatenverarbeitung auf einem solchen System ermöglicht. Dieses Framework besteht aus verschiedenen Modulen zur Hardwarebeschleunigung und bindet diese nahtlos in den Datenfluss der On-Board Software ein. Dabei wird außerdem die Möglichkeit genutzt, die eingesetzten *Virtex-4* FPGAs dynamisch zur Laufzeit zu rekonfigurieren.

Die Flexibilität des vorgestellten Frameworks wird anhand mehrerer Fallbeispiele aus der Bildverarbeitung von *PHI* dargestellt. Das Framework wird bezüglich der Verarbeitungsgeschwindigkeit und Energieeffizienz analysiert.

Abstract

Scientific instruments as payload of current space missions are often equipped with high-resolution sensors. Thereby, especially camera-based instruments produce a vast amount of data. To obtain the desired scientific information, this data usually is processed on ground. Due to the high distance of missions within the solar system, the data rate for downlink to the ground station is strictly limited. The volume of scientific relevant data is usually less compared to the obtained raw data. Therefore, processing already has to be carried out on-board the spacecraft.
An example of such an instrument is the *Polarimetric and Helioseismic Imager* (*PHI*) on-board *Solar Orbiter*. For acquisition, storage and processing of images, the instrument is equipped with a Data Processing Module (DPM). It makes use of heterogeneous computing based on a dedicated *LEON3* processor in combination with two reconfigurable *Xilinx Virtex-4* Field-Programmable Gate Arrays (FPGAs).
The thesis will provide an overview of the available space-grade processing components (processors and FPGAs) which fulfill the requirements of deep-space missions. It also presents existing processing platforms which are based upon a heterogeneous system combining processors and FPGAs. This also includes the DPM of the *PHI* instrument, whose architecture will be introduced in detail. As core contribution of this thesis, a framework will be presented which enables high-performance image processing on such hardware-based systems while retaining software-like flexibility. This framework mainly consists of a variety of modules for hardware acceleration which are integrated seamlessly into the data flow of the on-board software. Supplementary, it makes extensive use of the dynamic in-flight reconfigurability of the used *Virtex-4* FPGAs.
The flexibility of the presented framework is proven by means of multiple examples from within the image processing of the *PHI* instrument. The framework is analyzed with respect to processing performance as well as power consumption.

Contents

List of Figures

List of Tables

1. Introduction and Motivation

Data Processing Units (DPUs) for instruments on scientific space missions constitute the interface between the instrument's sensor and the spacecraft's on-board computer. Typical data to be acquired include spectral data, environmental conditions like magnetic fields, radiation and very often image data. Especially high-resolution cameras acquire a vast amount of data while downlink capacity is often strictly limited by transmission power and distance. The data rate can vary $\mathrm{Gbit/s}$ for missions in low earth orbit down to a few $\mathrm{kbit/s}$ for deep space missions (e.g. New Horizons mission to Pluto, *NASA*). Therefore, the volume of the collected data has to be compressed before it is handed over to the spacecraft's on-board computer or mass memory and final transmission to ground. The purpose of a conventional instrument DPU is shown in figure 1.1.
As the volume of the scientific information of interest is usually lower than the size of the acquired raw data, high-resolution sensors increase the demand for processing the data already on-board the spacecraft.
Systems for space applications always have special requirements due to the harsh environment and long term mission lifetime. Size, Weight and Power

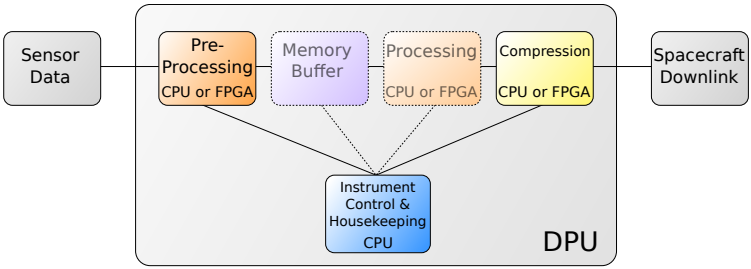


Figure 1.1.: Purpose of a data processing unit [1]

(SWAP) is highly constrained for such systems and therefore strictly limited. Power consumption is typically critical due to the restriction of solar power. Thermal restrictions and a high environmental temperature range are a limiting factor to the processing performance. While the cruise phase is dependent on the mission and can reach from several months up to a decade (e.g. for the Rosetta mission, *European Space Agency* (*ESA*)), also the operational phase of space missions is typically aimed for several years. This not only requires a conservative and robust hardware design of the DPU, but also mission objectives might change during this long period of time and the processing has to be adapted. All these requirements are a limiting factor to the choice of suited, available processing components. This typically includes General-Purpose Processors (GPPs) as well as Field-Programmable Gate Arrays (FPGAs).
One example for an instrument demanding for sophisticated on-board processing is the *Polarimetric and Helioseismic Imager* (*PHI*) on *Solar Orbiter*. Thus, the DPU of the *PHI* instrument is equipped with state-of-the-art processing components. This includes two in-flight reconfigurable, SRAM-based FPGAs for hardware acceleration. Nevertheless, these devices usually require a dedicated logic design. Therefore, the general need for a flexible, high-performance processing framework has still to be solved.

1.1. The Solar Orbiter PHI Instrument

A sophisticated example for an instrument on an upcoming space mission is the *Polarimetric and Helioseismic Imager* on-board *ESA*'s *Solar Orbiter* mission to be launched in 2020 [2].
The mission will observe the inner heliosphere of the sun and is planned to have an elliptical orbit with an aphelion of $0.9\,\mathrm{au}$ and a perihelion of $0.28\,\mathrm{au}$. The period of the spacecraft's orbit will be 150 days. Around the closest approach the orbit will be nearly sun-synchronous for a certain time. Therefore, particular spots of the solar heliosphere can be observed over a period of several days. This allows the mission to study the buildup of magnetic activity, e.g. solar flares. A general overview of the spacecraft and its instrument payload is given in figure 1.2.

PHI is a camera instrument providing high-resolution and full-disk measurements of the solar photosphere. The instrument will provide maps of the continuum intensity, the magnetic field vector and the Line-of-Sight velocity in the solar photosphere. Therefore, image data is acquired by an Active Pixel Sensor (APS) with a resolution of 2048×2048. During nominal operation, images are acquired at 4 different polarization states and at 6 different spectral positions, each. This results in a total of 24 images per data set. The pixel sensor is read out with a pixel depth of $12\,\mathrm{bit}$. To increase the Signal-to-Noise Ratio (SNR), multiple images will be accumulated during image acquisition, resulting in a maximum pixel depth of $22\,\mathrm{bit}$. Assuming a lossless compression (required for appropriate processing on ground) of factor two, the resulting size of one data set is $422.4\,\mathrm{Mbit}$.

The mission turns out to have a number of technical challenges, including the limited telemetry rate[1] to ground and the rough environmental conditions, both strongly dependent on the spacecraft's current position on the elliptic orbit [4]. Environmental conditions include thermal requirements as well as irradiation in case of solar activity.

These technical challenges apply to all of the components of the spacecraft and its instruments and specially have an impact on the data processing and the involved hardware. While on the one hand, thermal conditions limit the power consumption of electronic systems and solar activity demands a robust implementation against irradiation, the limited telemetry rate, on the other hand, requires a high reduction of the acquired data without the loss of relevant information. This only can be achieved by pushing classical ground processing steps to be performed already on-board the spacecraft. The extraction of the scientific parameters of interest can reduce the vast amount of data drastically but also demands for a high amount of processing performance.

The requirement for high-performance on-board processing, the needed flexibility to react for changing mission objectives and the limited availability of space-grade components offering the required performance and reliability not only demand for a sophisticated DPU. They also demand for a flexible processing framework which copes with the limitations of the resulting and underlying hardware architecture.

[1] The allocated telemetry rate for *PHI* is $20\,\mathrm{Kibit/s}$ for 3×10 days per orbit [3]

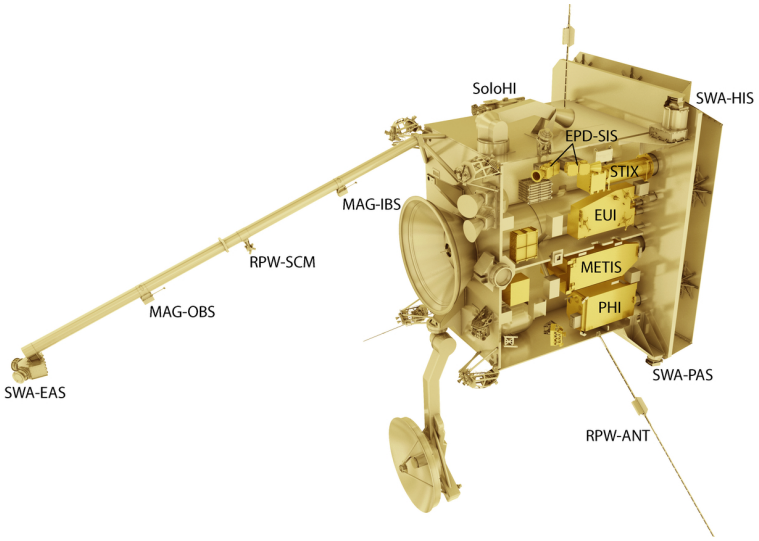


Figure 1.2.: Instruments on-board of Solar Orbiter [5]

1.2. Objectives and Contributions

The main objective of this work is to establish an image processing framework for sophisticated spaceborne camera instruments. This thesis covers the requirement analysis, technical challenges, implementation, testing and evaluation of such a framework. Although this is done exemplary for the *Solar Orbiter PHI* instrument, the underlying architecture and the resulting framework are applicable to other missions and instruments. This thesis is divided into the following parts:
A general overview into the field of spaceborne data processing modules is given in chapter 2. This covers an overview of traditional DPU architectures in section 2.2 and depicts a wide range of space-grade processing components which are currently available in section 2.3. This includes GPPs as well as FPGAs. After a short introduction of the basic effects of irradiation in space environments and its mitigation in section 2.4, related work in terms of

existing approaches for flexible and reconfigurable processing modules will be presented in section 2.5. The chapter ends with a detailed introduction of the *Solar Orbiter PHI* Data Processing Module (DPM) (section 2.6), which is the underlying platform for the processing framework and its components.
The actual flexible image processing framework is presented in chapter 3. After introducing related approaches of hardware accelerators and frameworks in the commercial field in section 3.1, the basic operation of *PHI* (data acquisition and processing) and the derived requirements are depicted in sections 3.2 and 3.3. The general architecture of the framework is presented in section 3.4. After presenting an overview of the overall implemented processing modules and giving an insight into the implementation of an excerpt of these modules in section 3.6, the chapter ends with a short introduction of the verification of the processing framework which is done using a *Python*-based framework for automated testing.
Chapter 4 will prove the concept of the presented framework by means of three different, exemplary parts from within the processing of the *PHI* instrument.
Finally, the presented processing framework is evaluated in chapter 5. This is done in terms of processing performance (section 5.1), power consumption (5.2) and FPGA resources (5.3). The thesis is summarized in chapter 6.

For this thesis, the following major contributions were made for the DPM of the *Solar Orbiter PHI* instrument:

1. The actual FPGA framework and software interface for the flexible on-board image processing. It consists of multiple modules which can be compiled into a set of FPGA configuration bitstreams. The framework is based on dynamic reconfiguration and is designed for (but not restricted to) the used *Xilinx Virtex-4* FPGA. It is implemented using *Very large scale Hardware Description Language (VHDL)* as well as *C* for the software interface running on the CPU (Central Processing Unit). It also includes an SDRAM (Synchronous Dynamic Random Access Memory) memory controller with a multi-port interface.

2. An automated test framework for verification and rapid evaluation of more complex processing procedures. The framework is implemented in *Python*.

3. A dedicated FPGA design for image acquisition and accumulation via a *ChannelLink* interface. This design also includes the SDRAM memory controller with a simplified multi-port interface.

4. A fault-tolerant, $512\,\mathrm{GiB}$ NAND-Flash mass-memory board and controller. The memory controller is implemented in *VHDL* and placed into a rad-hard by design *Microsemi RTAX2000* FPGA.

2. Data Processing Modules for Scientific Space Instruments

2.1. Challenges of Scientific Space Missions

Scientific space missions have to cope with a lot of different challenges. Each instrument on-board a scientific mission has its special purpose and science goals. It is intended to acquire and collect a particular set of measured data. In addition, each mission has its own constraints and requirements. This begins with a specific form factor of the housing for each component, in dependency of allowed volume and size. Each instrument has to fulfill its assigned mass budget. Depending on the spacecraft's orbit, there exist different constraints on the power consumption and thermal household. The maximum data rate for up- and downlink will vary with the spacecraft's distance to ground. Also requirements concerning radiation and mission lifetime are very mission-specific. Therefore, each instrument typically can be considered as a unique prototype.
Especially a limited down-link capacity, which is shared among a variety of different instruments, is often a bottleneck. The progressive development of high-resolution sensors and increasing volume of acquired data demands for more complex compression algorithms and on-board processing. Exploration missions to distant celestial bodies like Mars or future missions to asteroids will increase the role of space-robotics and additionally cope with the high latency caused by the long transmission path. Therefore, the need for autonomous robotics will also increase the demand for on-board processing capacity.
Technology for space-grade data processing is typically some years behind developments for the consumer market and has to be hardened to withstand the harsh space environment. The following sections will introduce the architectures of conventional data processing units followed by an overview of space-grade processing components available on the market.

2.2. Conventional Architectures

Traditional DPU architectures are typically based on space-grade, radiation-hardened components. For instrument control and spacecraft interface the use of at least one general purpose processor is mandatory. This can have the form of an Application Specific Integrated Circuit (ASIC) or be implemented as a System-on-Chip (SoC) inside an FPGA. Radiation-hardened ASICs as well as space-grade, typically one-time programmable FPGAs offer a quite low integration density. While ASIC based processor systems usually offer a little more processing performance, these devices only include a limited range of standardized interfaces for data acquisition. FPGA based processor systems, on the other hand, can be extended to specific interfaces and can include specialized processing cores. The downside is the lower integration density and operating frequency. Dependent on mission specific demands, a combination of both is often a suitable approach.
Good examples of conventional DPU architectures are the *Venus Monitoring Camera* (*VMC*) on-board *Venus Express* [6–8] and the *Dawn Framing Camera* on-board the *Dawn* mission [9]. Both DPUs are quite similar and make use of two Synchronous Random Access Memory (SRAM)-based *Xilinx QPro-R Virtex-I* FPGAs (*XQVR 600* and *XQVR 300*) integrating a *Gaisler LEON2* SoC with additional interfaces. Although these devices are SRAM-based, both the DPUs do not make use of in-flight reconfiguration. A one-time programmable *Actel SX32S* FPGA is used for configuration of the *Virtex* devices. Because of its radiation-hardness, it is also used as a watchdog controller as well as for Latch-Up protection. Both DPUs make use of the *RTEMS* real-time operating system. Figure 2.1 exemplary gives an overview of the *VMC* DPU architecture. The SoC has the following main features:

- LEON2 processor core, $20\,\mathrm{MIPS}$
- $1\,\mathrm{Gibit}$ image mass memory storage (SDRAM)
- SDRAM controller, $16\,\mathrm{Mbit/s}$ including DMA transfer
- $16\,\mathrm{Mibit}$ SRAM
- $64\,\mathrm{Kibit}$ PROM (Bootloader)
- $16\,\mathrm{Mibit}$ EEPROM (Program memory)
- Spacecraft Remote Terminal Unit (RTU) and *SpaceWire* interface
- Controller for Non-Volatile RAM (NVRAM)

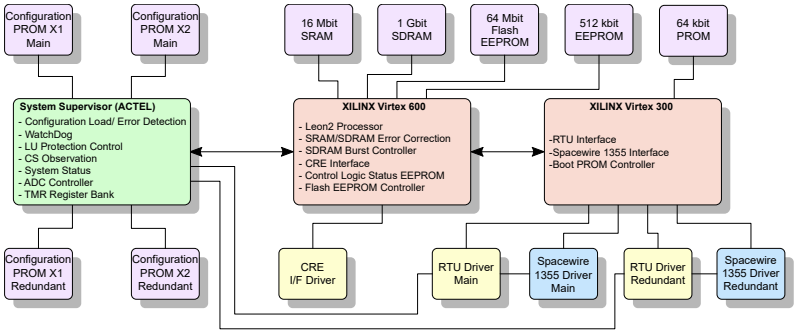


Figure 2.1.: Basic architecture of the *VMC* DPU

The *VMC* DPU's weight is about 280g while the average power consumption is about 3.2W. Another, relatively present example of a traditional DPU architecture is the DPU of the *JANUS* instrument of *ESA*'s *JUICE* mission [10, 11]. While it consists of a *LEON3* SoC in a dedicated ASIC, it includes two one-time programmable and radiation-hard *Microsemi RTAX2000* FPGAs for data acquisition, minimal pre-processing and standard CCSDS data compression.

2.3. Available Processing Components

For on-board processing tasks, a limited range of user-programmable digital processing components is available on the market. These components offer different levels of reliability and can be separated into general-purpose processors and FPGAs. The following sections will give an overview of the available parts.

2.3.1. Space-Grade General-Purpose Processors

Although early missions in the beginning of spaceflight in the early 1960s were carried out without general-purpose computers on-board, later missions including the Apollo program would not have been possible without the use of computers. Since this time an enormous progress in development of GPPs has

taken place. In 1965, Moore's Law predicted a doubling of transistors per chip every two years [12]. The time-line in figure 2.2 shows the development of space-grade processors in Europe and the United States from 1990 to now. The following subsections shortly summarize the the common processor families used in space applications.

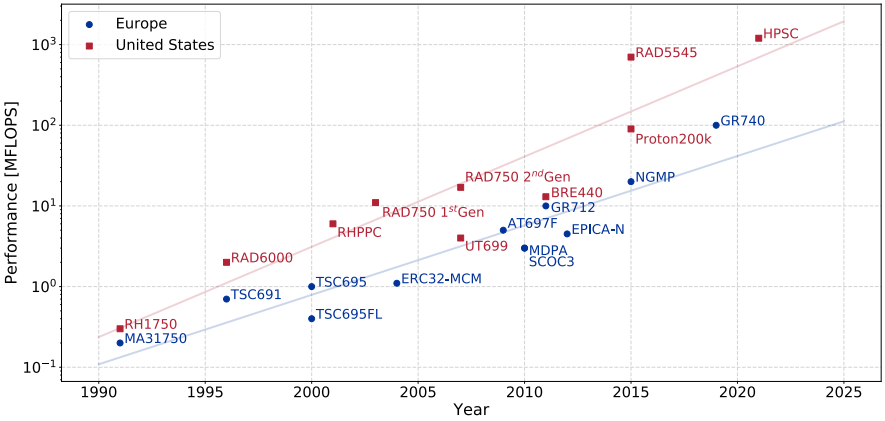


Figure 2.2.: Overview of general-purpose processors for space

Fault Tolerant LEON Microprocessor Family

The development of the 32-bit microprocessor based on the *SPARC-V8* Instruction Set Architecture (ISA) was initiated by *ESA* in 1997. The core is implemented in synthesizable VHDL. Meanwhile, the development is continued by *Gaisler Research*. It is available in the 4th generation and not only includes the processor core but a complete set of SoC components, including an AMBA bus architecture and peripheral cores. The LEON architecture is used in many European SoC designs. Besides a fault tolerant version, a public version is available under the GPL (GNU General Public License). The processor core provides the following key features:

- *SPARC-V8* instruction set architecture
- 7-stage pipeline
- Hardware multiply, divide and MAC (Multyply-Accumulate) units
- Fully pipelined double precision IEEE-754 Floating-Point Unit (FPU)
- Harvard architecture
- AMBA-2.0 AHB bus interface
- Symmetric Multi-Processor (SMP) support
- Fault-tolerant version for space applications
- $1.4\,\mathrm{DMIPS/MHz}$, $1.8\,\mathrm{CoreMark/MHz}$

One example of a System-on-Chip using the LEON3 architecture is the *GR712RC*, which was also developed by *Gaisler Research*. The chip is radiation-hard-by-design with contributions from *Gaisler* and *RadSafe* technology from *Ramon Chips*. It is based on a $180\,\mathrm{nm}$ *TowerJazz* CMOS process. It is qualified over the full military temperature range and for a Total Ionizing Dose (TID) of up to $300\,\mathrm{krad}$ and can be clocked up to $100\,\mathrm{MHz}$ (depending on external memory devices).
The SoC includes a fault-tolerant memory controller with configurable Error Detection And Correction (EDAC) supporting SRAM, SDRAM, Programmable Read-Only Memory (PROM), NOR-Flash and Input/Output (IO) address range. Furthermore a variety of interfaces is available via a configurable switch matrix, including six *SpaceWire* ports, Ethernet MAC, six UART ports, CAN-Bus interfaces and others. The *GR712RC* is used within the *PHI* DPU. Therefore, it will be covered in more detail in section 2.6.3.
An example using the *LEON4* architecture is the *GR740*, which is the successor of the *GR712RC*. In contrast to the *GR712RC* it includes four *LEON4* cores and operates at a clock frequency of up to $250\,\mathrm{MHz}$. The SDRAM main memory interface has a width of $64\,\mathrm{bit}$ ($96\,\mathrm{bit}$ with EDAC), which is twice as wide as for the *GR712RC*. A block diagram of the *GR740* processor is given in figure 2.3. Flight parts are expected to be available as of early 2019.

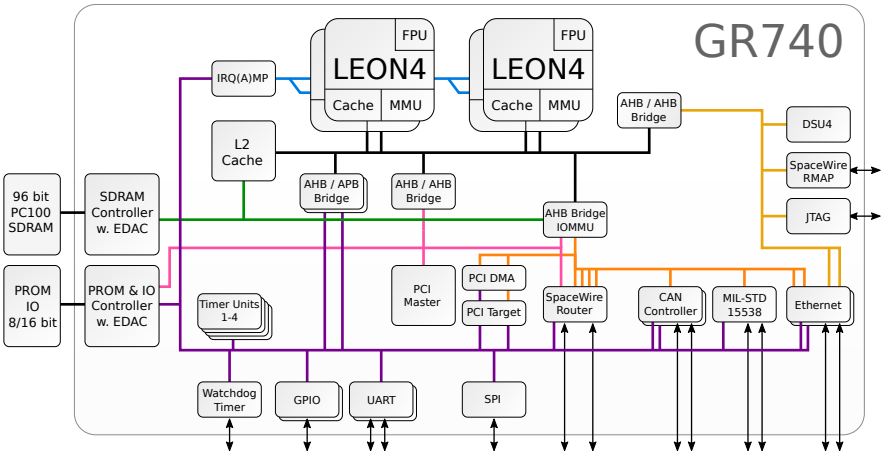


Figure 2.3.: GR740 SoC block diagram (simplified, according to [13])

RAD750 and RAD5545 PowerPC

The *RAD750* processor is based on the *IBM PowerPC 750* and is manufactured by *BAE Systems*. The processor runs with up to 200 MHz with a performance of ca. 400 DMIPS (Dhrystone MIPS) [14]. The radiation-hard version has a specified TID of up to 1 Mrad. The processor is designed to be used together with *Power PCI* companion ASIC to provide a bridge between the PCI backplane and the main memory [15, 16]. The first units have been launched in 2005 on *NASA*'s *Mars Reconnaissance Orbiter* and *Deep Impact* missions.
The successor of the *RAD750* is the *RAD5545* which is also from *BAE Systems*. The chip is based on a 45 nm Silicon on Insulator (SOI) process and the rad-hard by design *RH45* library. It has a performance of up to 5300 DMIPS. As well as the *RAD750*, it has a specified TID of up to 1 Mrad. It provides the following interfaces:

- 32 bit PCI interface
- Four serial RapidIO ports
- 16 *SpaceWire*link with embedded router
- Dual DDR2/DDR3 DRAM interfaces with interleaving
- NAND-Flash controller

RISC-V

A relatively new development in the domain of GPPs is the *RISC-V* ISA [17]. The architecture aims to a wide range of uses. It supports three different word-widths (32, 64 and 128 bit) which makes the ISA suited for a variety of systems, including embedded systems, personal computers, mobile devices as well as high-performance computing clusters. Therefore, the implementations can be classified in different subsets:

I: minimal set of integer instructions
M: singed and unsigned multiply and divide instructions
F: includes single precision floating-point unit
D: includes double precision floating-point unit
Q: includes quad precision floating-point unit
C: supports compressed instruction set format
E: only half the amount of 32-bit integer registers for use in embedded systems

More subsets are planned for bit-manipulation (B), decimal floating-point (L), packed Single Instruction, Multiple Data (SIMD) (P), vector processing (V) and transactional memory (T).
During the time of this work, no space-grade processor ASIC based on the *RISC-V* ISA is available [18]. Nevertheless the architecture is interesting for future space applications since *Microsemi* has introduced a *RISC-V* RV32IM IP-core to be used on the space-grade *RTG4* FPGA (see chapter 2.3.2).

Summary

In contrast to the commercial market, the availability of space-grade processors is very limited and the market is mainly dominated by *LEON* and *PowerPC* architectures. The processors which had been introduced in this section are summarized in table 2.1. All the shown proprietary processor implementations integrate commonly used, dedicated interfaces for communication with the spacecraft. This includes *MIL-STD-1553*, *SpaceWire* as well as Ethernet communication. As some GPPs provide relatively high processing performance, they might also have a high power consumption (e.g. the *RAD5545*). A quite new development, the *RISC-V* ISA, is not yet available as a dedicated, space-grade processor ASICs. As the ISA is an open standard, it might contribute in reducing costs for specialized designs and, thus, might play a key role in future developments.

Table 2.1.: Comparison of available rad-hard GPPs for space

	GR712RC [19]	**GR740** [13]	**RAD750** [14–16]	**RAD5545** [20, 21]
Processor Cores	2× LEON3 *SPARC V8*	4× LEON4 *SPARC V8*	1× *PowerPC*	4× *PowerPC*
Max. Clock Frequency	100 MHz[1]	250 MHz	200 MHz	n.s.
Performance	200 DMIPS	459 DMIPS per core	400 DMIPS	5300 DMIPS
Cache Size	2 × 8 KiB L1	2 × 8 KiB, 2 MiB L2	2 KiB L1, 1 MiB L2 [2]	2 × 32 KiB L1, 512 KiB L2, 2 × 1 MiB L3
IEEE-754 FPU	✓	✓	✓	✓
Memory Interface	32 bit SDRAM + EDAC	64 bit SDRAM + EDAC	64 bit SDRAM + EDAC [3]	64 bit DDR2/3 SDRAM, dual interleaved
Bus	IO-bus	33 MHz PCI 2.3	33 MHz PCI 2.2 [2]	PCI, *RapidIO*
SpaceWire Interface	6×	8× (integrated router)	-	16× (integrated router)
Ethernet MAC	10/100 Mbit/s	10/100/1000 Mbit/s	-	
Power Consumption	15 mW/MHz	not yet specified	1.5 W	17.7 W at 95 °C
Availability of FM Parts	since 2012	expected in 2019	since 2002	

[1] Max. 80MHz supported when using SDRAM memory
[2] L2 cache optional and off-chip
[3] Via companion ASIC

2.3.2. Space-Grade Field-Programmable Gate Arrays

Since many years FPGAs have been used for several space missions. These integrated circuits consist of an array of Configurable Logic Blockss (CLBs), enabling to program any logic circuit and functionality into the chip after the manufacturing process. Besides CLBs, modern FPGAs may include integrated memory (Block-RAMs (BRAMs)), elements for Digital Signal Processing (DSP), processor cores (e.g. ARM, PowerPC), phase locked loops, high-speed serial IO and much more.
There are several kinds of FPGAs available from different manufacturers. There are mainly three different FPGA technologies on the market:

Antifuse FPGAs This technology is based on a thin dielectric layer placed between two metalization layers. A permanent connection between the metal layers is established by applying a high programming voltage. The antifuse technique is non-volatile and the configuration of the device is resistant against Single-Event Upsets (SEUs). The user logic is operational immediately after power-up. Because the device configuration can not be read out, the technology is well-protected against reverse engineering. This is one reason why the technology is often used in military and aerospace applications.

Flash based FPGAs The configuration of the device is stored in Flash memory cells, thus the configuration is non-volatile and the device is functional immediately after power-up. Because Flash memory has less leakage compared to SRAM, FPGAs based on Flash technology have lower static power consumption compared to FPGAs based on SRAM technology. A main advantage is the possibility to reprogram the FPGA by a limited number of times (typically 500 programming cycles). So far, only the announced *Microsemi RTG4* has support for in-flight reprogramming of the device.

SRAM based FPGAs This category of FPGAs is holding the device configuration in static RAM cells. Because the SRAM technology is volatile, the configuration bitstream has to be loaded from an external source. Thus, the FPGA is not operational immediately after power-up. The SRAM cells are sensitive to SEUs, so that scrubbing of the configuration memory and mitigation techniques in the design are mandatory. A high benefit of this technology is the possibility of unlimited dynamic (and also partial) in-flight reconfiguration of the device.

While modern, commercial devices are equipped with millions of configurable slices and Megabytes of on-chip memory blocks, there is only a limited choice of parts available for the harsh space environment. The following subsections give an overview of the available FPGAs.

Microsemi RTAX

The *Microsemi RTAX* family of devices is based on anti-fuse technology combined with Triple Modular Redundancy (TMR) on cell level. Therefore, the device is radiation-hard by design and only requires a few actions by the designer which mainly applies to user-configurable RAM-blocks. Because of the used anti-fuse technology, the part is only one-time programmable which limits the design to be fixed and carefully tested before assembly of the individual electronic components. The limited programming also leads to a complex design prototyping because typically commercial, reprogrammable equivalents (e.g. the *ProASIC3E*) have to be used during development.
While the large *RTAX4000* device offers a capacity of 4,000,000 system gates (500,000 equivalent ASIC gates) and 67.5 KiB of BRAM, the smaller but (in terms of price and for prototyping reasons[1]) more frequently used *RTAX2000* offers 2,000,000 system gates (equivalent to 250,000 ASIC gates) and 36 KiB of BRAM.

[1] There is no prototype version of the *RTAX4000* available, which leads to disproportional high development costs.

Microsemi ProASIC3E

The *Microsemi ProASIC3* family of FPGAs is based on reprogrammable, non-volatile Flash technology. It is available in a radiation tolerant mil-grade version and, therefore, SEUs have to be mitigated by the user design. The device contains up to 3,000,000 system gates, $63\,\mathrm{KiB}$ of BRAM and up to 620 user IOs. Even though the device is reprogrammable, an in-flight reconfiguration of the user design is not officially supported by *Microsemi*.

Microsemi RTG4[2]

The *RTG4* is the fourth generation of radiation-tolerant FPGAs from *Microsemi* based on Flash-technology [22, 23]. As shown in figure 2.5, logic elements on the FPGA fabric are protected against radiation effects by using self-corrected TMR (STMR) flip-flops and Single-Event Transient (SET) filters, which can be optionally enabled to eliminate SET glitches up to $600\,\mathrm{ps}$.
The *RTG4* offers a variety of on-chip memory blocks. This includes 209 blocks of $24.5\,\mathrm{Kibit}$ dual-port SRAM and 210 blocks of smaller $1.5\,\mathrm{Kibit}$ three-port SRAM. In addition, the device has $374\mathrm{Kibit}$ of Flash-PROM for storage of e.g. IP-core configuration or coefficients.
For applications with focus on signal processing, the FPGA fabric includes 462 DSP math-blocks, which can operate on a maximum clock frequency of up to $300\,\mathrm{MHz}$. Two high-speed memory controllers are available as hard-IP cores on the device, used to connect external DDR2/DDR3 memory with up to $333\,\mathrm{MHz}$. These controllers also integrate Single-Error Correction and Double-Error Detection (SECDED) EDAC for upset mitigation. The device offers 24 lanes of high-speed Serialization/Deserialization (SERDES) with a data rate of up to $3.125\,\mathrm{Gbit/s}$. Especially for space applications, the fabric includes 16 circuits for *SpaceWire* clock and data recovery. An overview of the features included in the FPGA architecture is given in figure 2.4.

[2]The component was released in 2015 when design architecture considerations for the *PHI* DPU were already taken. It is presented for completeness.

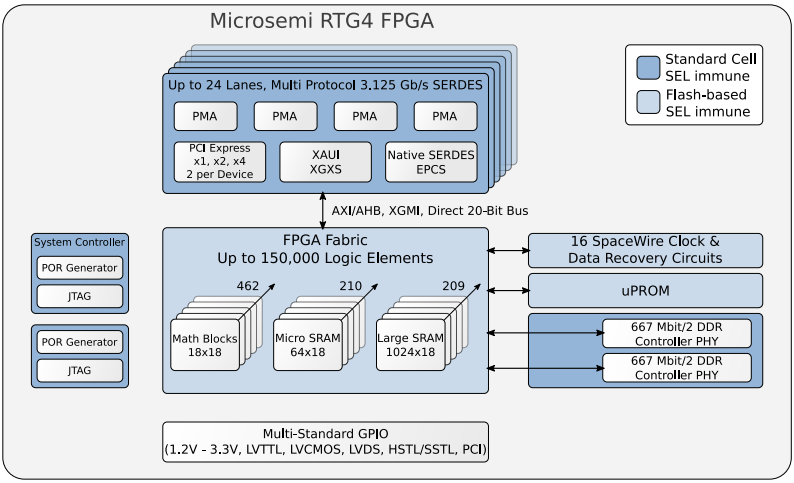


Figure 2.4.: Architecture of the new RTG4 FPGA, according to [22]

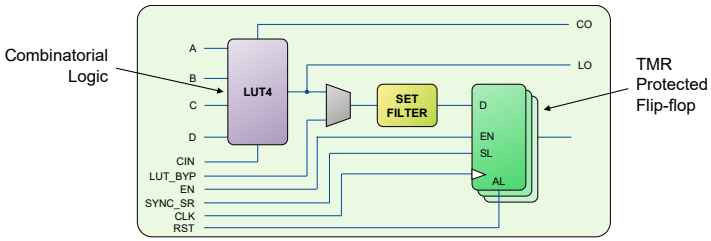


Figure 2.5.: RTG4 Logic Element (LE) [22]

Xilinx Virtex-4

In addition to *Microsemi*, the market leader *Xilinx* also offers a set of space-grade FPGAs. This includes the successful *Virtex-4* series, which is available in a qualified and pin-compatible Ceramic Column-Grid Array (CCGA) package. The radiation-tolerant devices are guaranteed for Total Ionizing Dose and Single-Event Latch-Up immunity. As their commercial equivalents, the *Virtex-4QV* FPGAs are offered in the following variations:

LX optimized for designs with high grade of custom logic
SX optimized for digital signal-processing using *DSP48* math blocks
FX optimized for processing including *PowerPC* hard-ip cores

These variations are possible through *Xilinx*'s Advanced Silicon Modular Block (ASMBL™) architecture. The structure contains multiple elements, such as Block-RAM, DSP slices, logic (CLBs) and IO buffers arranged into columns which can be placed in different quantities, depending on the FPGA variation. The structure of the *Virtex-4* specific Configurable Logic Blockss (CLBs) and the subdivision into logic slices is shown in figure 2.6. The *Virtex-4QV* family has a performance of up to $350\,\mathrm{MHz}$ and a TID of to $300\,\mathrm{krad}$.

Xilinx Virtex-5

With the *Virtex-5QV*, *Xilinx* also offers the successor of the space-grade *Virtex-4*. The only available space-grade device of the *Virtex-5* family is the *FX130*, which offers a total of 131,072 logic cells [25]. In contrast to its predecessor, the *Virtex-5QV* is fully implemented in TMR by design. It offers a performance of up to $450\,\mathrm{MHz}$ and a TID of $1\,\mathrm{Mrad}$.

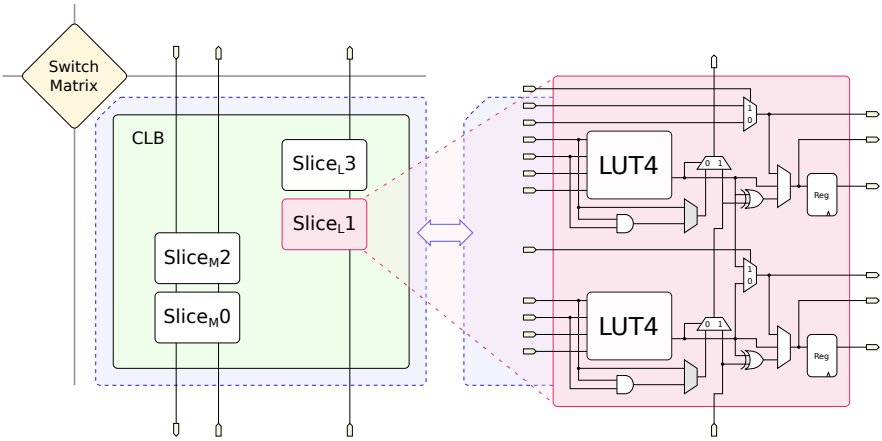


Figure 2.6.: *Virtex-4* CLB structure, according to [24]

Atmel ATF280F and ATF697FF

As a European alternative source for radiation-hardened technology, *Atmel France Aerospace* developed a set of ITAR-free (International Traffic in Arms Regulations) devices. This led to a set of SRAM-based FPGAs, with the *ATF280F* based on an $18\,\mu m$ CMOS (Complementary Metal-Oxide-Semiconductor) process as the latest development.
The *ATF280F* offers 280,000 ASIC equivalent gates at a maximum clock frequency of $50\,MHz$, which classifies to the low end of available parts in terms of capacity and performance. The device is radiation-hard by design using TMR. It overall contains $14\,KiB$ of user-configurable RAM-blocks. The SRAM-technology allows a reconfiguration of the device during operation.
Along with the *ATF280F*, *Atmel* offers a dual-package device including FPGA fabric (*ATF280*) and a LEON-2 based System-on-Chip (*AT697F*). The two components are connected through a PCI bus and are sharing General-Purpose Input/Outputs (GPIOs) and a memory interface.

NanoXplore NX1H35

NanoXplore is quite new in the field of space-grade FPGAs. Prototypes of the medium scale *NX1H35* will be available by end of 2017. The FPGA is based on SRAM technology with the advantage of automatic integrated scrubbing of the configuration memory. The fabric is done on a $65\mathrm{nm}$ process from *ST* which offers about 4400000 equivalent system gates with a TID of $100\ \mathrm{krad}$ [26]. The FPGA's application layer is already mitigated against single event effects by a combination of various techniques (e.g. TMR, EDAC). The architecture makes use of 4-input look-up tables and includes dedicated Digital Signal Processing (DSP) blocks and integrated Block-RAM. The device includes integrated *SpaceWire* interfaces. Besides device configuration via JTAG, SPI or an 8/16 bit parallel interface, the bitstream can also be loaded via a *SpaceWire* ECSS-E-ST-50-12C compliant link.

Summary

In summary, FPGAs have been widely used in space electronics for some time. They are essential for implementation of custom interfaces and glue logic in low quantities. Today's FPGAs integrate a high amount of configurable user logic, memory elements, IO interfaces and DSP elements. Space-grade devices are usually behind the commercial market. Meanwhile, the latest generations are re-programmable (either during development or even in-flight) and may contain specialized blocks such as *SpaceWire* clock recovery. As the inclusion of processor cores is still reserved for commercial parts, the space-grade devices are often used in combination with a dedicated General-Purpose Processor.

Table 2.2.: Available space-grade FPGAs

FPGA Component	Technology	Radiation hard	Radiation tol.	Logic Capacity	Max Freq. MHz	BRAM No. Blocks	BRAM Kibit	DSP Blocks
Microsemi								
ProAsic3EL RT3PE600L	Flash		✓	600,000[1]	350	24	108	–
ProAsic3EL RT3PE3000L	Flash		✓	3,000,000[1]	350	112	504	–
RTAX2000D	Antifuse	✓		2,000,000[1]	80	64	288	64
RTAX4000D	Antifuse	✓		4,000,000[1]	80	120	540	120
RTG4 RT4G150	Flash	✓		151,824	300	419	5,436	462
Xilinx								
Virtex-4 XQR4VSX55	SRAM		✓	55,296	400	320	5,760	512
Virtex-4 XQR4VFX60	SRAM		✓	56,880	400	232	4,176	128
Virtex-4 XQR4VFX140	SRAM		✓	142,128	400	552	9,936	192
Virtex-4 XQR4VLX200	SRAM		✓	200,448	400	336	6,048	96
Virtex-5 XQR5VFX130	SRAM	✓		131,072	450	298	10,728	320
Atmel								
ATF280F	SRAM	✓		280,000[2]	50	–	–	–
NanoXplore								
NX1H35	SRAM	✓		4,400,000[1]	n.s.	56	2,688	112

[1]Equivalent system gates
[2]Equivalent ASIC gates

2.4. Space Environment and Fault Mitigation

For the design of electronic systems on-board space instruments, special considerations for the harsh space environment have to be taken into account. At the beginning of a mission, the instruments have to resist the vibrations and the high acceleration which occurs during the launch of the spacecraft. Subsequently, the system has to withstand the surrounding environment for the complete mission lifetime which can reach from several years up to a few decades. This environment includes the high vacuum of space as well as harsh temperature conditions. The conditions for the on-board electronics are usually within a temperature range from $-30\,^{\circ}\mathrm{C}$ to $+125\,^{\circ}\mathrm{C}$. Depending on the purpose and operating conditions of a specific unit, this range might be less critical. E.g. for the *PHI* electronics, the non-operational temperature range is between $-30\,^{\circ}\mathrm{C}$ and $+60\,^{\circ}\mathrm{C}$.
While on the one hand this extreme temperature range has direct impact on electronics (e.g. propagation delay in digital circuits), on the other hand, temperature cycles will lead to strong mechanical stress and therefore, has an impact on the lifetime of components.
While on earth electronics are naturally protected against radiation from space, they are exposed to a number of sources outside the Earth's protective magnetic field. While this section only gives a short overview of the effects of radiation on digital integrated circuits (especially on SRAM-based FPGAs) and its fault mitigation techniques, a more detailed introduction to particle radiation in space environments can be found in [27, 28].

2.4.1. Radiation Effects on FPGAs

Radiation has a major impact on microelectronics and can either lead to temporary damage (soft error) or permanent damage (hard error) of a device. Effects can be distinguished between long term effects which accumulate over time by the influence of many particles or single effects which are caused by high-energetic particles.
Generally, the radiation effects can be categorized into the following types:

Total Ionizing Dose (TID) is a cumulative effect which comprises the total amount of ionizing radiation that a device absorbed over time. If a circuit has been exposed to radiation for a long term, undesirable effects may be observed. This includes an increased current leakage which leads to a higher power consumption and changes in the transistor threshold level. This, in turn, will affect the circuit's timing behavior. When in a proton-rich environment, the effects of TID can be lowered by shielding.

Single-Event Effects (SEEs) are short current spikes which are induces by particles traversing the device. The effects strongly depend on the energy of the particle, the location of the strike and the conditions to that time. The strike of a heavy ion into the area of a pn-junction will create free electron-hole pairs. Under the effect of an adjacent electric field these pairs will separate which leads to a short current spike if a critical charge is reached (see figure 2.7). The effects of this current spike can be furthermore classified into:

Single-Event Transient (SET) in which a gate that is part of combinational logic is affected by a transient pulse. The resulting glitch may spread along the combinational path and might get captured by a flip-flop which then leads to an SEU. When hitting a global reset line, an SET might cause an accidental reset. Also, a transient on a global clock line can lead to faulty sampled signals and metastability effects. In general, synchronous designs are less sensitive to SETs than asynchronous designs. SETs can be reduced using a glitch filter.

Single-Event Upset (SEU) which occurs when the transient pulse of an SET affects a storage element (flip-flop or DRAM cell) directly, or gets sampled by a clock edge after propagation through a combinational path. One single event affecting more than one memory cell results in a Multiple Cell Upset (MCU). If the upsets are within a single memory word, it is referred to as a Multiple Bit Upset (MBU). Countermeasures include Error Detection And Correction (EDAC) (especially for memories) and TMR.

Single-Event Functional Interrupt (SEFI) occurs when a part of the integrated circuit with control functionality is hit by an SEU which puts the system into a faulty state. For example, this can be a stuck Finite-State Machine (FSM) or an altered, invalid counter value. Typically, the functionality can be recovered by a reset or power-cycle.

Single-Event Latch-Up (SEL) which in contrast to the previously characterized effects can lead to a permanent damage of the device. The effect is triggered when a particle strike ignites a parasitic thyristor inside a CMOS structure, resulting in a short circuit between supply voltage and ground. If not rapidly stopped by power cycling, this can lead to thermal destruction. The sensitivity of a device can be drastically reduced during manufacturing process, i.a. by using Silicon on Insulator (SOI) technology (which is also done for the space-grade *Virtex-4 QV*).

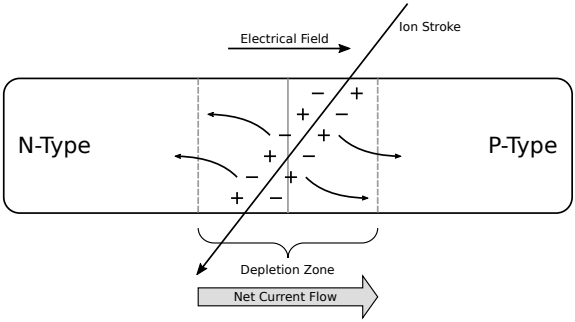
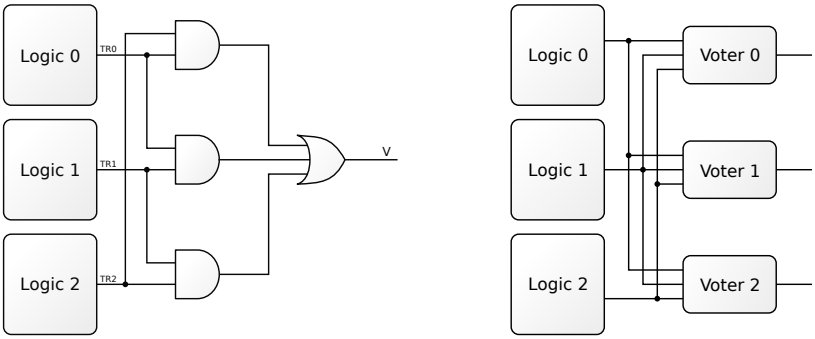


Figure 2.7.: Ion strike into a pn-junction [29]

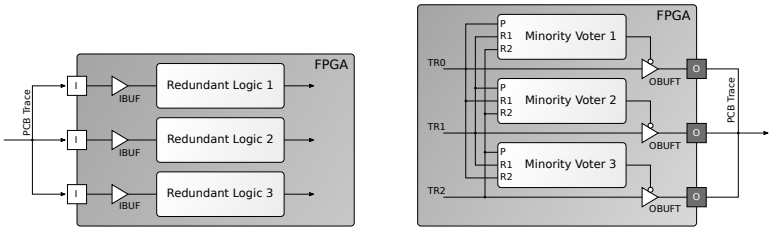
2.4.2. Fault Mitigation Techniques for SRAM FPGAs

When regarding the fault model of SRAM-based FPGAs, two different layers have to be taken into account: the configuration layer and the application layer. The configuration layer consists of SRAM cells holding the information about the configuration of the device. This can be global routing information, configuration of CLBs or DSPs, contents of Look-Up Tables (LUTs), or other configuration relevant information.
The application layer represents all the specific circuitry and memory elements for the actual user design. Upsets in these elements, for example, will affect the user data or corrupt finite state machines. The ratio of user flip-flops to the relevant configuration memory cells of a *Virtex-4 SX55* FPGA is about 0.003% [27]. Exemplary, the estimated upset rates for the used *Virtex-4QV SX55* are shown in table 2.3 [30].
To deal with Single-Event Effects (SEEs) on the application layer of FPGAs, TMR in combination with integrated voters is commonly used. While a simple voter (as shown in fig. 2.8a) would be susceptible to SEEs, the voting mechanism itself has to be considered redundant to guarantee adequate mitigation (fig. 2.8b). The TMR mechanism may already be integrated into the technology on a fine-grain cell level, as done e.g. for the space-grade *Virtex-5* or the *RTG-4*. Another method is to consider module-based TMR on the design level. This is typically done by automated design tools after logic synthesis. In this case, no modifications of the technology on cell-level are needed. In case of the *Virtex-4*, several tools are available for the insertion of TMR. This includes the *XTMR* tool provided by *Xilinx* and the *BL-TMR* tool which is a research project from *Brigham Young University* and *Los Alamos National Laboratory* [31].
The insertion of TMR into the design is also able to cover IO interfaces. Figure 2.9 shows the coverage of IO pins on the example of the *XTMR* tool. Therefore, the related input or output pins are connected on the PCB (Printed Circuit Board). When applying TMR on output pins (figure 2.9b), a faulty signal output will be set to high-impedance.



(a) Simple voter (b) Redundant voter

Figure 2.8.: Different voting mechanisms



(a) Input from package pins (b) Output through package pins

Figure 2.9.: TMR through IO interfaces [32]

Table 2.3.: Estimated upset rates for *Xilinx Virtex-4QV SX55* [30]

	total upsets $\text{device}^{-1}\text{s}^{-1}$	**total upsets** $\text{bit}^{-1}\text{s}^{-1}$
solar minimum	5.4×10^{-5}	3.5×10^{-12}
solar maximum	1.9×10^{-5}	1.2×10^{-12}
worst week	0.88	5.7×10^{-8}
worst day	4.3	2.8×10^{-7}
worst 5 minutes	16	1.0×10^{-6}

2.5. Related Work

Since the flexible image processing framework strongly depends on FPGA technology in combination with dynamic reconfiguration, this section will cover a selection of available space-grade processing modules and applications which make use of reconfigurable FPGAs. These examples are partly commercial and partly originate from funded research.
Even though the flexible image processing framework developed in this thesis can be applied to a number of underlying hardware platforms, the Data Processing Module of the *Solar Orbiter PHI* instrument is used as an underlying hardware basis. Therefore it will be discussed in depth at the end of this section.

2.5.1. SEAKR Application Independent Processor Architecture

The *SEAKR Application Independent Processor Architecture* (*AIP*) is a commercial, modular platform [33] based on the *CompactPCI* standard. The 6 unit *CompactPCI* back plane accommodates the power supply (22-36 V), a *PowerPC*-based Single Board Computer (SBC) and a board including a reconfigurable FPGA. Furthermore, the system can be extended by one 6 unit spare PCI slot.
The integrated *Athena-3* SBC is based on a *PowerPC e500* core (32 bit), featuring 2450 DMIPS at 1 GHz clock frequency. The CPU core is connected via a 32 bit PCI bus which is clocked at 33 MHz. Furthermore, it is equipped with 1 GiB of DDR2 SDRAM working memory and 128 KiB of EEPROM. Additionally, it can be configured to contain 3 GiB of non-volatile Flash memory. The module includes *SpaceWire*, Gigabit Ethernet and *MIL-STD-1553* connectivity for interfacing other spacecraft components. Furthermore, the system offers 56 configurable GPIOs.
The reconfigurable FPGA processor board can include up to three *Xilinx Virtex-5 FX130T* FPGAs. Versions with a *Virtex-4 LX160* or *LX200* also seem to be available [3]. The FPGA module itself is equipped with up to 3 GiB of DDR2 SDRAM buffer memory. Configuration bitsreams are stored inside 16 Gbit

[3] As this commercial product seems to be partly confidential, no further details are available.

of configuration memory with optional, dedicated configuration scrubber. The module can be further extended by a custom mezzanine board.
The size of the platform is $30 \times 25 \times 20\,\mathrm{cm}$ (WDH), with a total weight of around $10\,\mathrm{kg}$. The overall power consumption is strongly dependent on the system configuration and FPGA usage. It is within the range of approximately $40\,\mathrm{W}$ to $80\,\mathrm{W}$.

2.5.2. Fraunhofer On-Board Processor (FOBP)

The Fraunhofer On-Board Processor (FOBP) platform evaluates the use of reconfigurable hardware to allow more effective telecommunication satellites and to test novel communication protocols. The FOBP is part of the research payload of the german Heinrich Hertz communication satellite *(H2Sat)* mission. In contrast to traditional telecommunication systems which use a classical bent pipe and therefore simply amplify and forward the received signal, the FOBP approach will completely process the received signal in the digital domain. This includes digital filtering, channel decoding/encoding and routing. The digital processing allows to improve the overall system performance by enabling error correction on the received signal (regenerative payload) or on-board routing.
To achieve the performance needs for digital signal processing of the signal, the on-board processor platform makes use of four reconfigurable, space-grade *Xilinx Virtex-5* (*FX130*) FPGAs. The platform mainly consists of four functional units which include analog front ends, digital signal processing, a communication unit for Telemetry and Telecommand (TM/TC) handling and periphery (e.g. memory and clock) [34]. Figure 2.10 gives an overview of the FOBP architecture.

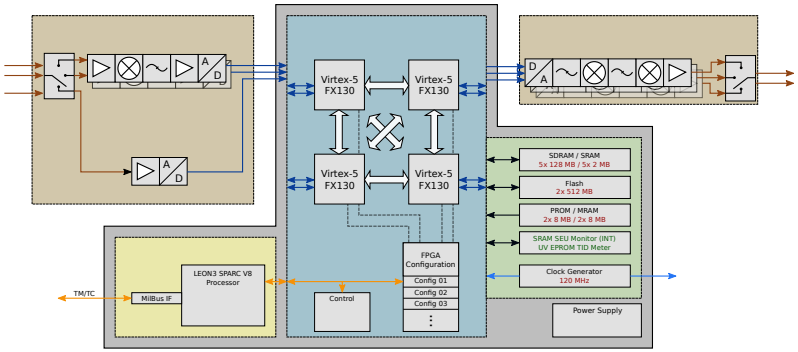


Figure 2.10.: Architecture of the Fraunhofer On-Board Processor (FOBP) [34]

2.5.3. The Dynamic Reconfigurable Processing Module (DRPM)

The Dynamically Reconfigurable Processing Module (DRPM) was developed at *IDA* as a research project in collaboration with *Airbus DS* and funded by *ESA* [35]. The main goal of this study was to build a demonstrator platform (shown in fig. 2.13) and evaluate the use of dynamic and partial reconfiguration, offered by modern SRAM-based FPGAs, for the space environment.
The minimum system configuration of the DRPM consists of a System Controller (SC), a *SpaceWire* Router and at least one Dynamic Reconfigurable FPGA Module (DFPGA). All modules are connected by the *SpaceWire* router. The system is designed to be modular and extensible so it can be expanded by further DFPGAs or connected to other DRPMs. Thus, it is possible to increase the system's processing capability as well as to add redundancy for safety critical designs. The overall DRPM design architecture is depicted in figure 2.11.
The superior System Controller serves as an interface to the spacecraft (S/C). It is the central configuration manager and system supervisor for the subordinate DFPGAs. The module is equipped with a *SpaceWire* Remote Terminal Controller (RTC, AT913E) device, which consists of a fault tolerant *LEON2* processor-system, attached to a NAND-Flash memory for storage of full and

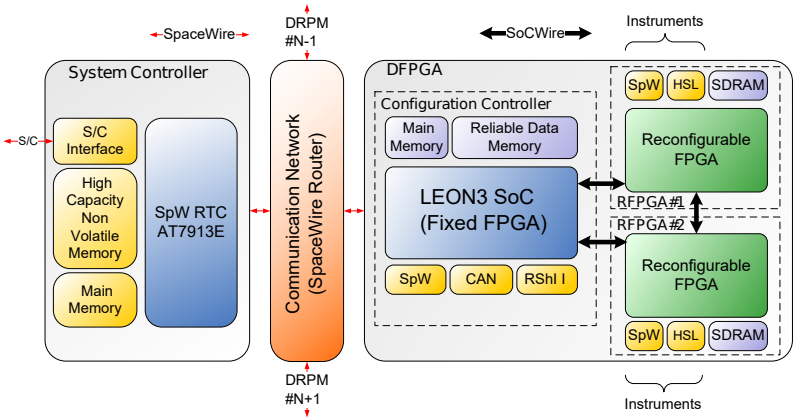


Figure 2.11.: Architecture of the DRPM [35]

partial FPGA configuration. The content of this memory is protected by a software-based data triplication while the system's working memory and non-volatile memories for application software storage are protected by EDAC mechanisms. In case of a failing non-volatile memory, the software can be uploaded directly to the working memory via the *SpaceWire* Remote Memory Access Protocol (RMAP).

The actual FPGA processing is carried out on the system's DFPGA module. It is divided into a Configuration Controller (CC) and two Reconfigurable FPGA modules (RFPGA), each located on a separate mezzanine board. The Configuration Controller is itself is based on a LEON3 System-on-Chip within a fixed FPGA, which is based on the commercial *GRLIB* IP library from *Cobham Gaisler*. It connects to the System Controller via the *SpaceWire* router on the one hand, and to the two Reconfigurable FPGAs (RFPGAs) on the other hand. The two RFPGAs are interfaced by the SoCWire protocol, a robust and fault tolerant Network-on-Chip (NoC) developed at *IDA* which is based on the *SpaceWire* standard. The CC also contains low-level interfaces such as SPI and CAN-Bus as well as general-purpose IO. A copy of the RFPGA configurations is stored inside the attached non-volatile, EDAC-protected NOR-Flash

memory. The (re)configuration and also the background scrubbing of the connected SRAM-based FPGAs is done by *SelectMap* interfaces connected via the *SoCWire* NoC. Each of the RFPGA slots can be equipped with a mezzanine board containing either an SRAM-based *Xilinx Virtex-4 SX55* or *FX140* FPGA, but can be easily adapted to any other technology or future FPGA types. The detailed architecture of the DFPGA Configuration Controller is depicted in figure 2.12.
While the DRPM demonstrator is fully equipped with commercial parts (except the *SpaceWire* RTC), these devices can be easily changed to its space-qualified equivalents.

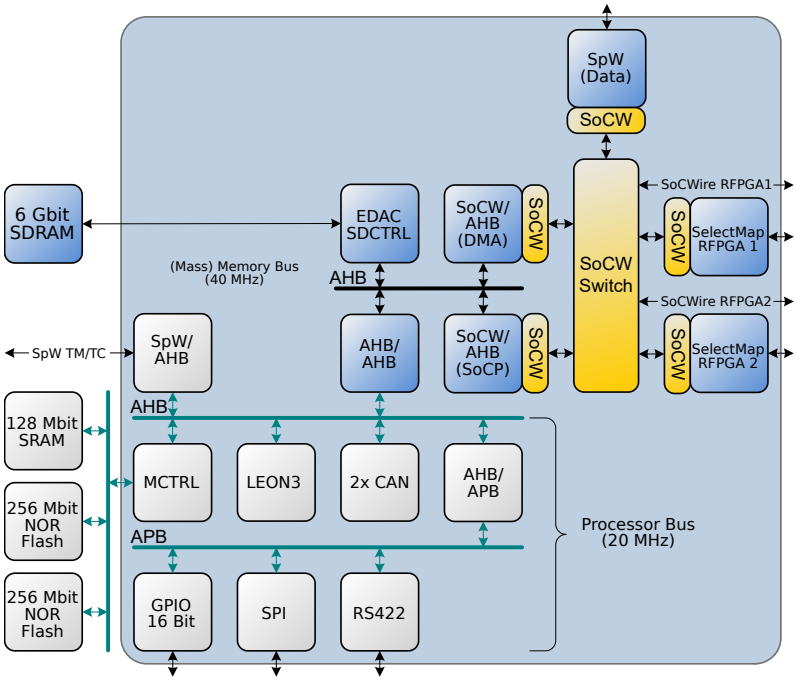


Figure 2.12.: Detailed architecture of the DFPGA Configuration Controller [35]

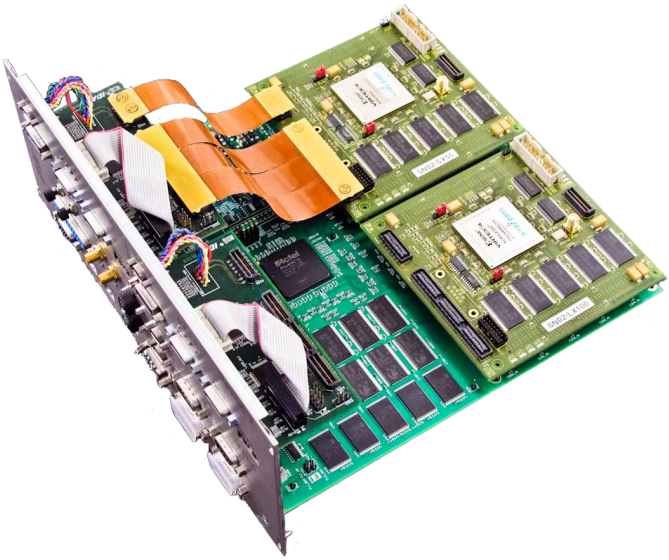


Figure 2.13.: DRPM hardware platform

2.6. Solar Orbiter PHI DPM

Like the processing modules presented in the previous chapter, the DPM for the *Solar Orbiter PHI* instrument is using reconfigurable FPGA technology. In this case, two SRAM-based *Xilinx Virtex-4QV SX55* FPGAs are integrated into a subsystem containing a radiation hardened general purpose processor and a supervisor FPGA. The hardware architecture is derived from the DRPM study. Since this work is mainly based on the *PHI* DPM [1], the following subsections will cover the overall instrument and especially the architecture of the processing module in detail.

Figure 2.14.: *PHI* DPU Qualification Model (without NAND-Flash module)

2.6.1. The Polarimetric and Helioseismic Imager

As already introduced in section 1.1, the *Polarimetric and Helioseismic Imager* is a camera instrument on-board the *Solar Orbiter* mission. Observing the inner heliosphere of the Sun, it has the following scientific main objectives [3]:

1. How does the solar dynamo work and drive connections between the Sun and the heliosphere?
2. What drives the solar wind and where does the coronal magnetic field originate from?
3. How do solar transients drive heliospheric variability?
4. How do solar eruptions produce energetic particle radiation that fills the heliosphere?

PHI will acquire high-resolution and full disk measurements of the solar photosphere which are used to provide maps of the continuum intensity, the magnetic field vector and the Line-of-Sight velocity within the solar photosphere. Therefore, the instrument will scan six different wavelength positions with each four different polarization states around the $\mathrm{FeI}-6173\,\text{Å}$ line. This results in overall 24 images with a resolution of 2048×2048 pixels each. To improve the SNR, multiple images are accumulated during image acquisition, extending the bit depth from $12\,\mathrm{bit}$ per pixel to a maximum of $22\,\mathrm{bit}$. The instrument is equipped with a Full-Disc Telescope (FDT) as well as a High-Resolution Telescope (HRT). While the FDT has a Field-of-View (FOV) of $7208.96\,\mathrm{arcsec}^2$, the HRT only has an FOV of $16.8\,\mathrm{arcmin}^2$. An example for the expected results after successful on-board processing is shown in figure 2.15.

Figure 2.16 shows the main components of the instrument as a functional diagram, mainly consisting of an Optics Unit (OPT) and an Electronics Unit (ELE). The Optics Unit is divided into the two separate optical paths for FDT and HRT, each containing its own Polarization Modulator (PMP) and re-focus mechanism. Unlike the FDT, the HRT comes with an image stabilization which consists of Correlation Tracker Camera (CTC) and a tip-tilt mirror/drive. As FDT and HRT can be used alternatively, the optical path can be switched by a feed select mechanism. Both channels use a common filtergraph and Focal

[1] The *PHI* DPM is sometimes internally also referred as *PHI* Data Processing Unit (DPU)

Plane Assembly (FPA). Latter one includes the actual pixel sensor and its front-end electronics.
All the subsystems of the instrument are supplied with power from and controlled by the Electronics Unit. It comprises two main and redundant PCM, a High-Voltage Power Supply (HVPS) for the filtergraph, the Tip-Tilt Controller (TTC) for the image stabilization subsystem as well as the Analog, Mechanism and Heater Drivers (AMHD) and the DPU with its mezzanine memory module.

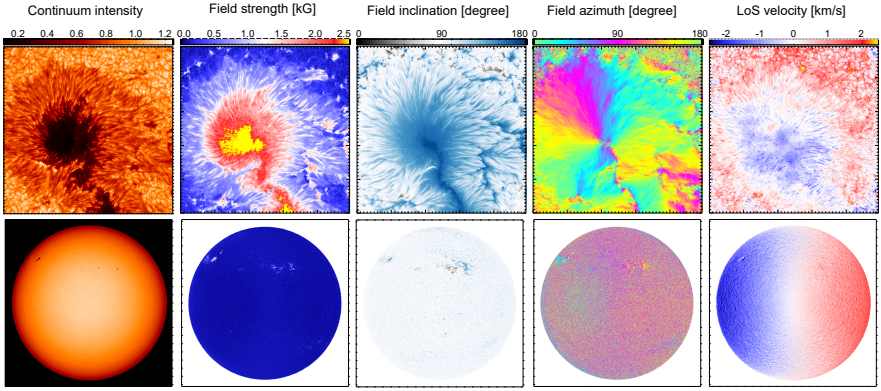


Figure 2.15.: Data products resulting from *PHI*'s on-board processing [3]

As the DPU controls and monitors all the instrument's subsystems, it can be considered as the central element of *PHI*. Especially the sophisticated needs for data storage and on-board processing make high demands on the hardware of the DPU.

The following two sections will cover the general requirements and the derivative system architecture of the *PHI* DPU in detail.

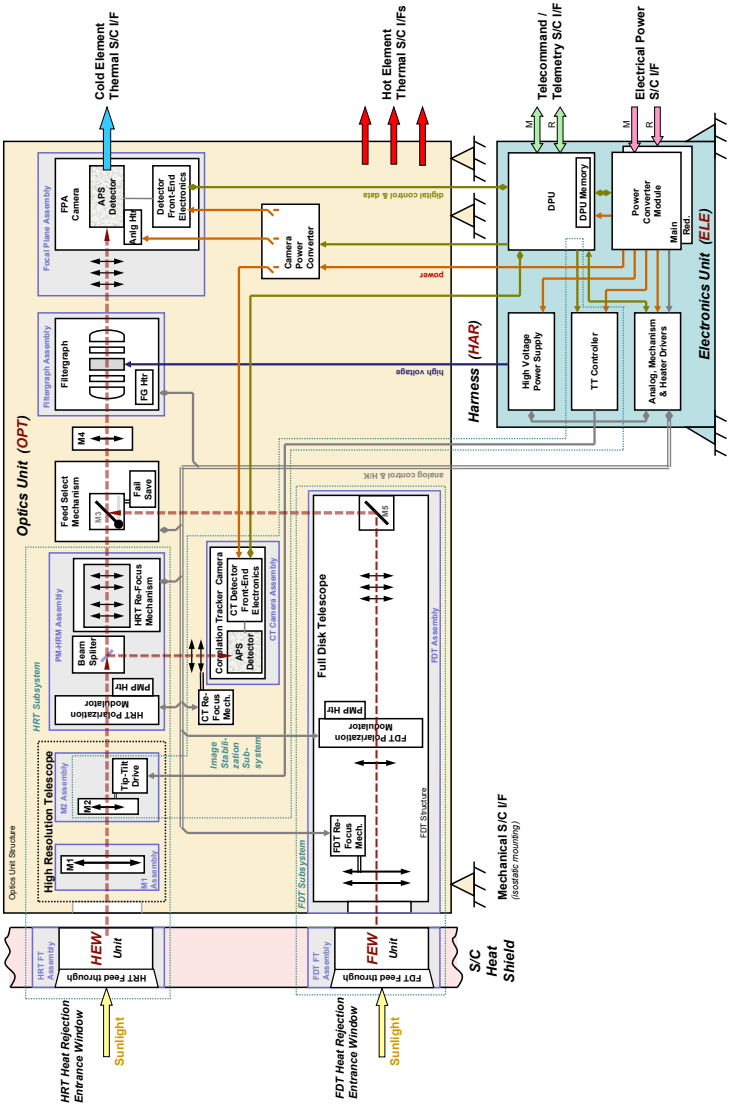


Figure 2.16.: *PHI* Functional Diagram [36]

2.6.2. Architecture Overview

The specific on-board data processing requires a set of dedicated real-time function cores to be implemented in hardware within the FPGAs. To allow for real-time processing of the very demanding Radiative Transfer Equation (RTE) inversion algorithm, the implementation has to be massively parallelized and would need devices with huge logic resources, e.g. *Virtex-4 FX140* or *Virtex-5 FX130*. Certainly, such a completely parallel real-time processing is not achievable for the instrument, mainly due to limited resource utilization (power and heat dissipation). However, full real-time processing and data acquisition is not mandatory, not all of the functional modules need to operate concurrently if intermediate data buffering is implemented. An unused module resident in an FPGA wastes resources. It would be sufficient if a functional module could be requested to be instantiated and run in an FPGA on demand. The ability of SRAM-based FPGAs to support dynamic (partial) reconfiguration allows this flexible use of the available hardware platform in a Time-Space Partitioning (TSP) manner even for complex algorithms [37–39].
The dynamic reconfiguration approach of the *PHI* DPM enables multiple use of the FPGA resources during different modes of operation which will be used for both, simple update capability of hardware functions during the long mission and dynamical in-flight reconfiguration for sharing of complex algorithms on limited FPGA resources. The operational and processing requirements of the *PHI* instrument greatly facilitate the Time-Space Partitioning of the four different modules at two different operation modes:

- One dedicated configuration using two FPGAs for image acquisition:
  - Image Stabilization System (ISS) in RFPGA 1
  - Data accumulation in RFPGA 2
- Second, different configuration for the subsequent data processing:
  - RTE inversion in RFPGA 1
  - Data pre-processing in RFPGA 2

The basic architecture of the *PHI* DPM design is based on the results of the *ESA* study for a Dynamically Reconfigurable Processing Module (DRPM) which has been presented in section 2.5.3. The modular DRPM is based on *SpaceWire* communication, a General-Purpose Processor and two reconfigurable SRAM FPGAs. Based on this, a design dedicated to the *PHI* DPM was

derived. The complete high-level architecture of the *PHI* processing module is shown in figure 2.17. It utilizes a combination of a powerful *Cobham Gaisler GR712RC* processor ASIC including two fault tolerant *LEON-3FT* cores together with a fixed, radiation hardened and TMR by design, one-time programmable *Microsemi RTAX* FPGA as system supervisor plus a set of dedicated hardware function cores implemented within in-flight reconfigurable *Xilinx Virtex-4* FPGAs. A combination of small amount of volatile buffer memory (8 Gibit SDRAM) and large capacity of non-volatile image memory (4 Tibit NAND-Flash) provides significant storage capacity, which fulfills all needs of intermediate data storage at very low resources. The design of the NAND-Flash based system will have complete error correction, taking into account the Flash handling [40]. NAND-Flash based mass storage for space have been intensively studied by our group in the Safe Guard Data Recorder (SGDR) study for *ESA* and have been already implemented for the Sentinel-2 SSMM (Solid-State Mass Memory) [41] and ExoMars PDHU. Communication between the CPU, the NAND-Flash controller and the RFPGAs including their submodules is established using the *SoCWire* protocol, a robust and fault-tolerant Network-on-Chip (NoC) implementation in dependance on the*SpaceWire*standard [42]. While the *GR712* CPU and the system supervisor on the *RTAX* FPGA are running at a clock frequency of 50 MHz, both RFPGAs are supplied with an external clock of 100 MHz.

Table 2.4.: Overview of different memories used for the *PHI* DPU

Memory	**Purpose**	**Technology**	**Capacity**	
	System Controller			
PROM	Bootloader	non-volatile	32	KiB
NOR-Flash	Software, FPGA Bitsreams, FAT	non-volatile	128	MiB
SDRAM	Working Memory	volatile	256	MiB
	RFPGA 2			
SDRAM	Buffer Memory	volatile	1	GiB
	System Supervisor			
NAND-Flash	Mass Storage	non-volatile	512	GiB

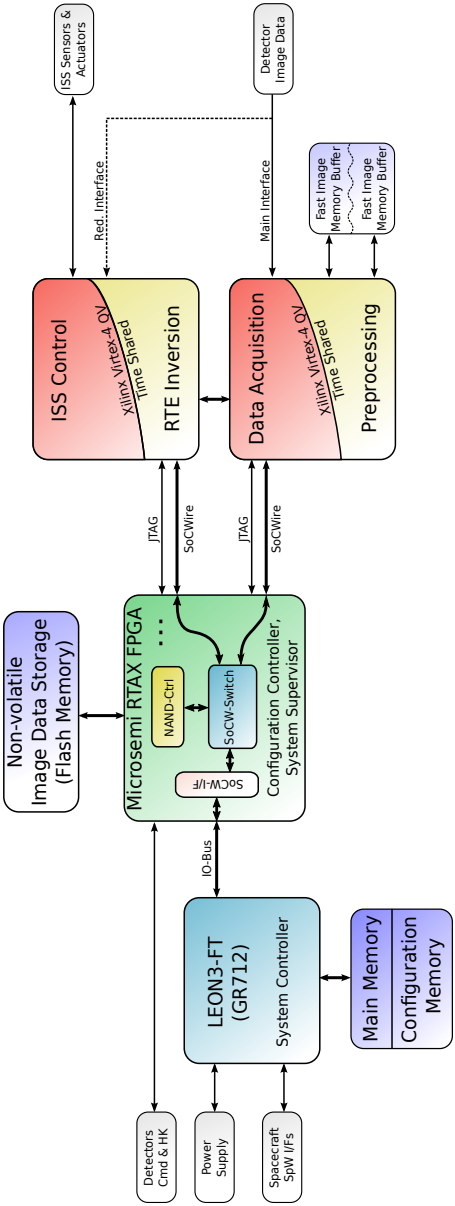


Figure 2.17.: Architecture of the *PHI* DPU

2.6.3. System Controller

The System Controller connects the *PHI* DPM with the spacecraft's On-Board Computer (OBC) and SSMM via two redundant *SpaceWire* interfaces. It is responsible for TM/TC of the spacecraft as well as controlling and monitoring all the sub components of the instrument. Therefore, this essential element of the DPM has to be highly reliable. For this reason, the radiation hardened and fault tolerant *Cobham Gaisler GR712RC* processor ASIC has been selected for this purpose.
The *GR712RC* SoC includes two *LEON3FT* cores (see section 2.3.1) which run with a clock frequency of up to 100 MHz. It supports a variety of interfaces including CAN, Ethernet, SPI, I^2C, UART, SLINK and *MIL-STD-1553*. Figure 2.18 gives a brief overview.
The processor allows to connect 32 bit wide external SDRAM memory to the system including 16 bit of Reed-Solomon EDAC. In case of the *PHI* DPM, 256 MiB of external memory are connected to the System Controller. The System Supervisor in the *RTAX2000* FPGA and therefore the two RFPGAs are interfaced via the processor's IO memory interface. The use of external SDRAM and the connection via the IO interface slows down the maximum operation frequency of the processor. Thus, the *GR712* of the *PHI* DPM is only clocked with a frequency of 50 MHz.

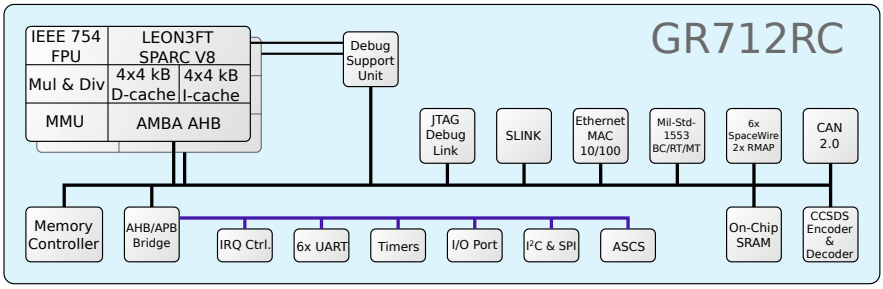


Figure 2.18.: Block diagram of the *GR712RC* used as System Controller [19]

2.6.4. NAND-Flash Controller and Memory Board

In order to allow offline-processing of the accumulated images which have been acquired during observation phase and therefore enable a time-shared operation of the processing hardware, the *PHI* DPM needs a vast amount of non-volatile storage capacity. An acquisition of one complete data set of 24 observables à 8 MiB results in 192 MiB of data (an acquisition of 10 data sets would already result in $\approx$ 1.9 GiB). This demand for capacity can only be satisfied by the use of NAND-Flash memory which is widely used in commercial devices (e.g. mobile phones, tablets and Solid-State Drvies).
A variety of commercial Single-Level Cell (SLC) NAND-Flash devices have been tested under irradiation [43, 44] and found to be suited for the use in the *PHI* instrument. These selected commercial *Micron* 128 Gibit devices (*MT29F128G08AJAAA*, [45]) have been repackaged and stacked by *3D-Plus*. Overall 48 stacked commercial devices (24 memory stacks) have been placed on an additional mezzanine board, including related circuitry (figure 2.19).

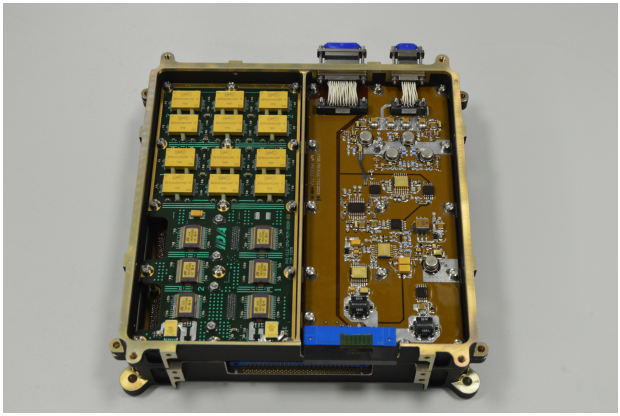


Figure 2.19.: Half-sized mezzanine NAND-Flash memory board (left)

NAND-Flash Memory Organization

Because the *GR712* CPU does not include a dedicated interface to connect overall 48 NAND-Flash devices, a special controller has to be integrated into the System Supervisor FPGA [40, 46]. The used *RTAX2000* FPGA only offers a low number of IO pins. Therefore, the physical connection of overall 48 independent memory devices to the System Supervisor requires additional glue logic.
To achieve redundancy, the NAND-Flash module is organized in two redundant memory partitions which can be powered individually. Furthermore, each memory device inside the stacks contains two targets with separate control signals (e.g. chip enable). This results in overall 48 independent targets. Six targets are combined to 16 slices, resulting in a data bus which is physically $48\ \mathrm{bit}$ wide. To implement an efficient *Reed-Solomon* error correction, two independent slices are forming a Word Group (WG). These slices theoretically could also be from different partitions. To operate only one partition at a time, the two slices are restricted to be from within the same partition. This comprises 12 devices each, resulting in a logical bus width of $96\ \mathrm{bit}$.
The data bus is isolated by a set of drivers for each partition. All control signals which are needed for the asynchronous interface of the memory devices are distributed and isolated by a set of latches which allow to set and hold the control signals for each slice individually.
Figure 2.20 shows the organization of all the comprised NAND-Flash devices with the use of driver and latch components. The overall memory board has a net capacity of $512\ \mathrm{GiB}$ ($768\ \mathrm{GiB}$ gross capacity including EDAC). An overview of the size of various sub units is given in table 2.5.
Even though no latch-ups have been observed during radiation testing, the current of each partitions is monitored and will be switched off immediately in case of high current flow. This protects the devices against burn-out. The monitoring is done by comparing the voltage against a fixed reference over a shunt resistor.

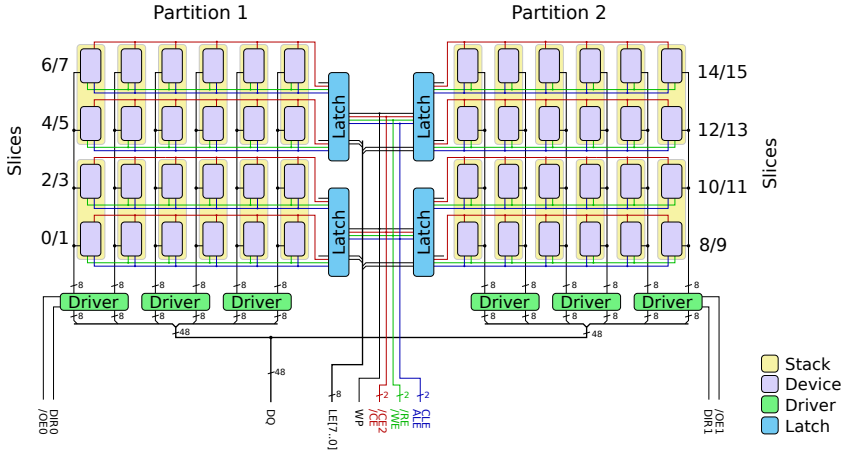


Figure 2.20.: Organization of NAND-Flash Devices on the Memory Board

Table 2.5.: Capacity of NAND-Flash memory devices and board

Page Size			8	KiB
Block Size	128	Pages	1024	KiB
Target Size	8192	Blocks	8	GiB
Device Size	16384	Blocks	16	GiB
Stack Size			32	GiB
Total Board net Capacity			512	GiB
Minimal writeable/erasable Unit			8	MiB

NAND-Flash Memory Controller

To connect the overall 48 NAND-Flash devices to the *GR712* CPU and the two RFPGAs, a dedicated controller has been implemented inside the *RTAX2000* system supervisor FPGA. As mentioned in the previous section, the NAND-Flash controller makes use of the asynchronous interface of the *Micron* devices. It supports the following key features:

- Asynchronous NAND-Flash interface operation
- *SoCWire* interface for main data transfer
- Two-symbol Reed-Solomon error detection and correction
- Direct access to multiple and single memory devices via CPU IO interface
- Interrupt driven
- 96 bit logical data bus (64 bit net)
- 48 bit physical data bus (32 bit net)
- Data rate: $\approx 90\ \mathrm{Mbit/s}$ write, $\approx 40\ \mathrm{Mbit/s}$ read
- Current limiting and switch-off
- Auto power on/off and write protection

The simplified architecture of the NAND-Flash controller is depicted in figure 2.21. It shows the three used main data paths:

Asynchronous NAND-Flash Interface This path is used for control and data transfer to/from the 48 memory devices. The interface comprises overall 48 bidirectional data lines, 7 control signals and 12 additional signals to control the latch/driver circuitry to multiplex the devices on the separate memory board.

***SoCWire* Interface** The *SoCWire* Network-on-Chip (see section 2.6.6) is the main path for read/write transfer of the actual user data. It connects the controller to the System Controller as well as the two RFPGAs. It includes the two symbol *Reed-Solomon* error correction (coder and decoder).

Control Interface This interface connects the memory controller to the System Controller (*GR712*) via the IO-memory interface of the CPU. It is mainly used for control purpose, e.g. reading/writing of status and control registers, commanding of the memory controller and transfer of addresses in the background while transfer via *SoCWire* is executed. It can also be used as a secondary data path to the memory from within the System Controller. This secondary data path bypasses the *Reed-Solomon* EDAC and can be used for debug and special purpose.

While data transfer is foreseen via the *SoCWire* interface (including EDAC), the control interface enables to fully access and command single memory devices from within the system software. This is mainly used for maintenance purpose, e.g. to obtain device unique ID, get/set features, read status, device reset and erasing of blocks/devices.

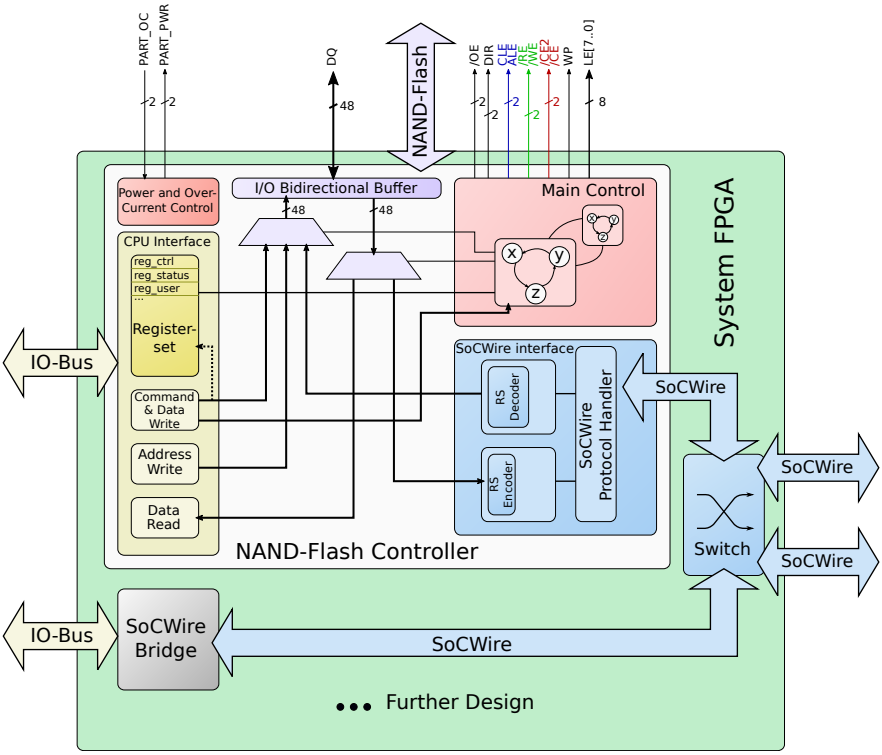


Figure 2.21.: Architecture of NAND-Flash Memory Controller

Specialized NAND-Flash File System

In order to safely operate the integrated 512 GiB NAND-Flash board, a simple file system has been implemented. It allows to load and store parameters, acquired raw image data as well as processed scientific data. The file system holds a table for bad-block management, which is initialized by reading out bad-block markings from each of the 96 NAND-Flash devices. Additional bad-block entries can be added manually.
The wear-leveling mechanism is kept very simple: each memory partition is completely filled with acquired or computed data. If no more space is available, the partition is cleaned up, erasing all blocks belonging to files which are marked as deleted.
The File Allocation Table (FAT) itself is not stored inside the NAND-Flash memory but in the much smaller NOR-Flash memory which also holds flight-software and FPGA configuration bitstreams. The FAT is handled in a fail-safe way, ensuring that file allocation information is written before the actual storage into NAND-Flash (vice versa during cleanup operation). As well as NAND-Flash technology, the NOR-Flash is also block-based and has a limited number of erase cycles. To assure lifetime of the FAT locations in the NOR-Flash memory, file system information is only altered by flipping memory cells from logic '1' to '0'. This can be done without erasure of blocks. Thus, blocks in the NOR-Flash memory are not erased more frequently than blocks in the NAND-Flash memory.

2.6.5. Reconfigurable FPGAs and Scrubbing

The main innovation of the design of the *PHI* DPM is the usage of the two reconfigurable *Xilinx Virtex-4QV SX55* FPGAs to significantly accelerate onboard data processing. These devices are used in a space-grade version based on a different Silicon on Insulator (SOI) process which is immune against latch-ups. Furthermore, the FPGAs are packed into a hermetically sealed CCGA package. Compared to its commercial equivalent which comes in a Ball-Grid Array package, the chip has nearly the same footprint (1140 instead of 1148 pins). To compensate the different thermal expansion of the ceramic

package compared to the commercial version and the PCB, the solder balls are replaced by columns. In order to use the devices on the *Solar Orbiter* mission, the assembly of the package had to be qualified according to *ECSS* standards.
In addition of the lower price of the devices compared to the also available *Virtex-5*, the *Virtex-4* was chosen for the *PHI* DPM because it has a lower static power consumption compared to its successor. The *Virtex-5* has the advantage that the application layer is protected with TMR by-design. For the *Virtex-4*, the designer has to add fault mitigation of the application layer manually to the design, supported by the *XTMR-Tool* which is provided by *Xilinx*.
Independent of the protection of the application layer, the correction of Single Event Upsets in the configuration-layer of the FPGA has to be done by scrubbing of the configuration memory. This has to be done for the *Virtex-4* as well as the *Virtex-5* FPGA.
For configuration and scrubbing, the *Virtex-4* has multiple interfaces (e.g. *SelectMap*, JTAG or internal ICAP). Due to the limited pin-count of the used *RTAX* System Supervisor FPGA, the *PHI* DPM uses the serial JTAG interface. The custom and lightweight JTAG controller is located in the System Supervisor FPGA, allowing independent access to each of the two RFPGAs. The configuration and scrubbing procedure is mainly controlled and executed by software. The configuration bitfiles are stored within the NOR-Flash memory where also the system software is stored. After configuration of a device, the configuration memory is continuously scrubbed by software in a background process. Areas, containing Block-RAM information will be omitted. For effective scrubbing of the FPGAs, the following three methods have been implemented in the software of the instrument [30, 47]:

Blind Scrubbing The content of the configuration memory is continuously overwritten by the actual design. A drawback of this method is that an upset in the configuration logic (the JTAG controller inside the *Virtex-4*) could lead to a complete misconfiguration of the FPGA. Also the CPU load of the *GR712* is quite high.

Readback Scrubbing The content of the configuration memory is read back frame-wise and compared against the original configuration bitfile and a mask. The advantage of this method is that only parts of the

configuration memory are scrubbed when an actual SEU in a specific region is detected. This reduces the probability of misconfiguration of the FPGA. The disadvantage is an even higher CPU load than the blind scrubbing method.

Readback Scrubbing with FrameECC Similar to the readback scrubbing the configuration memory is read back frame-wise. In contrast to the simple readback method, the JTAG controller dumps the data read from the JTAG interface. The ECC (Error-Correcting Code) of each frame is evaluated by an IP-core provided by *Xilinx* which is instantiated in the user design. The result is then fed back from the user design to the JTAG chain by a boundary scan logic component after each frame and is finally checked by software. This method has the advantage of the readback method with significantly reduced CPU load.

Because of the advantages of the reduced CPU load and the lower probability of device misconfiguration, the readback scrubbing with support of the FrameECC IP-core is the preferred solution. Unfortunately, the space-grade *Virtex-4 SX55* devices have issues with crosstalk in the configuration logic. This leads to the wrong detection of sporadic upsets at low environmental temperatures which have been observed during tests of the flight model. Thus, a combination of FrameECC readback scrubbing with a blind scrubbing interval after a specific number of readback cycles is finally used.

2.6.6. SoCWire Network-on-Chip

All the particular components of the system are spread across over all three different FPGAs. For reliable control and data transfer, all the modules of the *PHI* DPM are connected by a *SoCWire* Network-on-Chip. *SoCWire* has been developed as a prior research project at *IDA* [48–50] and was used for the DRPM study which was presented in section 2.5.3. The *SoCWire* protocol is influenced by *SpaceWire* [51], which is used as a reliable connection between different subsystems on spacecraft level.

The NoC approach transmits packets from source to destination of a point-to-point connection. These packets are routed via network fabric switches

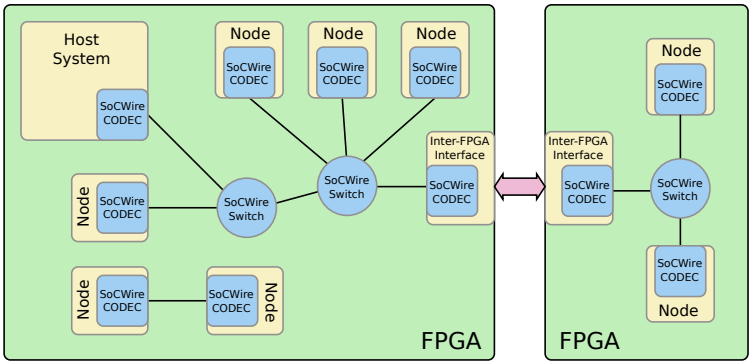
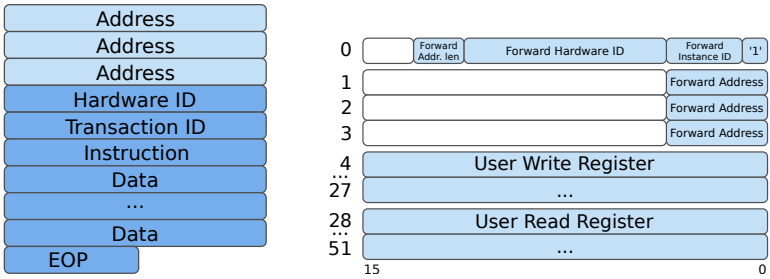


Figure 2.22.: Example of a SoCWire Network-on-Chip constellation [50]

(routers). Figure 2.22 shows some example use cases of a *SoCWire* NoC constellation. In the simplest case, *SoCWire* can just connect two nodes without a switch in between the two elements. A larger network comprises a host system and multiple nodes. These nodes can be connected through multiple switches. *SoCWire* allows a maximum of three switches in one path. To allow even an off-chip connection, an Inter-FPGA interface has been implemented specially for the *PHI* DPM. *SoCWire* is optimized for low resource utilization on FPGAs. Because it is also used within the system supervisor FPGA of the *PHI* DPM (*RTAX2000*), the implementation was slightly modified to get a more lightweight system.

In contrast to *SpaceWire* which uses serial data transfer with clock recovery, *SoCWire* is a fully synchronous protocol with parallel data transfer. The word width can be configured. For the *PHI* DPM, a word width of 16 is used. Packets are routed from source to destination node via switches. *SoCWire* uses path addressing which is limited to three addresses. Therefore, a packet can only be routed through a maximum of three switches. Each switch reads and removes the first forward address from the packet and routes the data with the left addresses to the specified output. The path to the controlling host processor is always defined by routing through port 0 of each switch. Figure 2.23a shows the general format of a *SoCWire* packet which consists of the three path addresses, the destination hardware ID and a transaction ID. The subsequent in-



(a) SoCWire Packet Format

(b) Register Set in a SoCWire Protocol Handler

Figure 2.23.: SoCWire Protocol, according to [52]

struction word differentiates between the modes listed in table 2.6 which allows streaming of data and read/write access to the user registers. The transmission of a packet ends with an End of Packet (EOP) character.

Table 2.6.: Different *SoCWire* packet instructions

Register write request	Indicates a write access to a user register of a node
Register write reply	Acknowledges a successful write access to a user register
Register read request	Indicates a read access from a user register of a node
Register read reply	Returns the register value after a successful read of a user register
Process request	Indicates a data packet to be processed by the core
Process reply	Returns a packet of data which was processed by the core
Stream packet	Indicates a streaming data packet

The register set of a *SoCWire* node is depicted in figure 2.23b. The first four registers are used to set the destination hardware/instance ID and routing path of the streaming data. These are typically set by the host system, allowing packets to be forwarded from one node to another. Besides these configuration registers, each node can have up to 24 read and 24 write user-registers. The *PHI* DPM uses a fully synchronous *VHDL* description to implement the *SoCWire* protocol inside the FPGAs.

2.6.7. Software

The *PHI* DPM's software is running on the *GR712* CPU and is written within the *C* programming language [53]. Besides the flight software which is contributed by *MPS*, an independent software is used for testing at *IDA*.
Both of these two variations are used on top of the bootloader which is stored in a separate Read-Only Memory (ROM) device. This bootloader allows to run an alternate software which can be loaded during start-up via the system's *SpaceWire* interface and the RMAP protocol [54]. Usually, if no other software is loaded during start-up, the bootloader copies the main software from the DPM's NOR-Flash into the SDRAM working memory. The NOR-Flash memory is divided into main and redundant partitions, so that the bootloader automatically will load the software from the redundant partition in case an error is detected.
While the main flight software includes all necessary routines to operate the complex instrument and to implement the compliant interface to the spacecraft's OBC and mass memory, the internal test software is a light-weight implementation for testing of the DPM's hardware and FPGA components. Nevertheless, both variants are running on top of the RTEMS *4.10* real time operating system [55] and share common parts. This comprises drivers and Serial Peripheral Interfaces (SPIs) for the various hardware and FPGA components, including:

- Extended GPIOs and serial interfaces on *RTAX* System Supervisor FPGA
- Driver for bridge to the *SoCWire* Network-on-Chip
- Driver for NAND-Flash memory controller and filesystem API
- Driver for NOR-Flash write operations and filesystem API
- Driver for JTAG configuration and API for background scrubbing of RFPGAs
- API functions for control of the processing flow

They both also include a compiler/interpreter system of the On-Board Command Language (OCL) [56, 57]. This provides a programming language mainly following the syntax of *C* which allows the safe implementation of procedures for the instrument. OCL was introduced and verified for the *OSIRIS* camera instrument on the *Rosetta* mission.

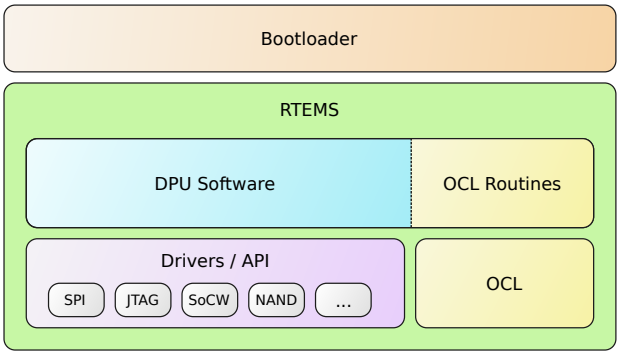


Figure 2.24.: General layers of software usage in *PHI* DPM

2.6.8. Summary

In summary, the *PHI* DPM comprises a radiation-hard *LEON3 GR712* processor ASIC in combination with two in-flight reconfigurable *Virtex-4* FPGAs for processing and data acquisition as well as 512 GiB of NAND-Flash memory for intermediate storage. All these components are interfaced by a one-time programmable and rad-hard by-design *RTAX2000* FPGA which includes the NAND-Flash memory controller, several IO interfaces and two JTAG configuration controllers. These controllers enable the dynamic configuration and continuous safe operation of the two SRAM FPGAs through software-based scrubbing. Communication between the several components is done via an specialized Network-on-Chip architecture, the *SoCWire* network. The on-board software makes use of the *RTEMS* real-time operating system and is stored in a dedicated NOR-Flash memory. An update of the software through *SpaceWire* is guaranteed by a bootloader located within a fixed PROM. The on-board software contains all the necessary drivers and API for operation of the hardware components. Furthermore, it includes a specialized On-Board Command Language (OCL) for execution of instrument procedures.

3. A Flexible Image Processing Framework

The DPM for the *Solar Orbiter PHI* instrument was already introduced in chapter 2.6. The previous chapter already presented a general hardware platform for the instrument's DPU which was derived from the DRPM study. As already mentioned, the hardware design offers the ability of in-flight FPGA reconfiguration. This enables the instrument to use its resources in a time-shared way which is ideally suited for the flow of the *PHI* instrument. While the previous chapter mainly concentrated on the general hardware design of the instrument's DPU, this chapter will present the general flow of the instrument's data processing including the data acquisition. After presenting approved commercial approaches for hardware acceleration and deviating the requirements for a flexible image processing pipeline, the base architecture of the framework is presented.

3.1. Related Commercial Approaches

Hardware acceleration is widely used in commercial applications, especially in the computer graphics domain. These Graphic Processing Units (GPUs), widely used from the beginning of the late 1990s, usually contain multiple specialized Processing Elements (PEs) used for manipulation of texture images and vertices. These GPUs further evolved to the field of general purpose computing. Nowadays, GPUs are used for the computation of a variety of problems. This particularly includes e.g. image processing, crypto mining, weather forecasting, machine learning and many more.
Besides GPUs, also FPGAs have a growing share in high performance computing. FPGA vendors like *Xilinx* or *Altera* provide frameworks for out-sourcing of computation intensive tasks to FPGA based hardware accelerator components. One approach for the standardization and abstraction of various hetero-

geneous hardware (either FPGA or GPU) and interfaces of different vendors is the *Open Computing Language* (*OpenCL*) [58], which was introduced in 2008 by the *Khronos Group*.
The platform model of *OpenCL* is depicted in figure 3.1, which divides the computing system into a host system and a number of compute devices. These devices are further divided into compute units and Processing Elements (PEs), which can be vendor specific. While the compute units and PEs contain small amount of buffer memory (e.g. for line buffering), each compute device is equipped with a larger, typically external, global memory. Each compute device can either be a CPU itself, a GPU or even an FPGA. These devices are typically located externally via a bus but could also be placed inside a single SoC (e.g. *ARM* SoC with GPU, *Xilinx Zynq* with programmable logic area).

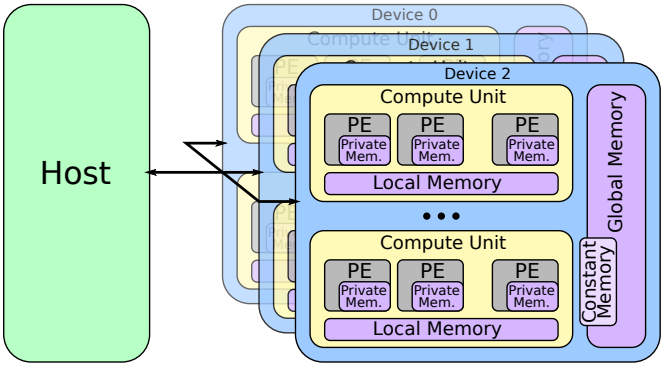


Figure 3.1.: *OpenCL* platform and memory model [59]

Computing intensive processing tasks which shall be accelerated are executed on the computing device. These tasks, the so called computing kernels, can either be *OpenCL* kernels or built-in kernels. *OpenCL* provides two methods for the description of *OpenCL* computing kernels, the *SPIR-V* intermediate language or *OpenCL C*, which is an extended subset of the *C* language. The kernels will be compiled just-in-time before execution on the computing device. In contrast to the *OpenCL* kernels, the build-in kernels are vendor specific. In case of an FPGA accelerator, these could be RTL code defined by a Hardware

Description Language (HDL) or dedicated IP-Cores.
To benefit from acceleration of the computing devices, data (usually vector or image data) has to be transferred to the memory of the computing device before execution of one or more kernels. After computation has finished, the result has to be copied back to the host system.

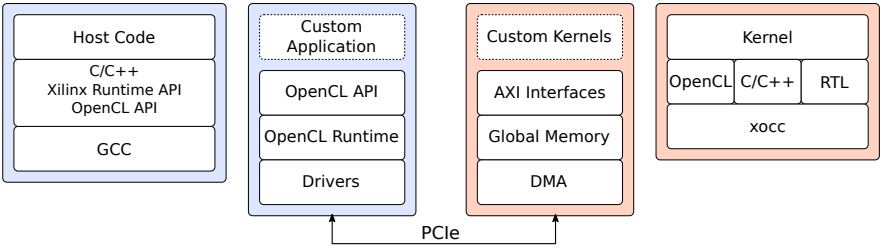


Figure 3.2.: *Xilinx SDAccel* platform and tool-chain [60]

FPGA based platforms for hardware acceleration using the *OpenCL* framework are offered by *Intel* (formerly *Altera*) and *Xilinx*. Because only *Xilinx* is targeting the space market, the *SDAccel* framework shall be shortly presented at this point. It can be used within a combination of a host PC and an FPGA system (e.g. using *Kintex UltraScale+* or *Virtex UltraScale+*) located on a PCIe card. Also the use inside a single System-on-Chip is possible when using the *Zynq UltraScale+* platform. Furthermore, the framework also targets for cloud based high-performance computing. Although it offers the use of *OpenCL* it also can be used independently. Figure 3.2 gives an overview of the platform and the corresponding tool-chain.

3.2. PHI Data Acquisition and Processing

The operation of the *PHI* instrument generally can be divided into two phases: The acquisition phase and the processing phase. During acquisition phase, typically when the spacecraft is in the perihelion of his orbit around the sun (ca. $0.28\ \mathrm{au}$), series of images with different filter settings will be acquired of the sun's surface and stored onto the DPU's NAND-Flash memory. For a better signal-to-noise ratio, up to 24 images can be accumulated.
Because of spacecraft jitter which occurs during the acquisition of the images, the instrument includes a tip-tilt image stabilization inside the optical path. To control the tip-tilt mirror, a real-time evaluation of images from low-resolution the tip-tilt camera is needed while the data acquisition and accumulation is running.
After the acquisition phase is over (typically when spacecraft is entering aphelion of the orbit), data is recalled from the NAND-Flash memory and processed to extract scientific parameters. Before the actual extraction of the magnetic field vectors is carried out by the inversion of the RTE, the image data has to be pre-processed.
As shown in table 3.1, these main four processing intensive tasks can be perfectly assigned to the two SRAM FPGAs in a time-shared manner:

Table 3.1.: FPGA task allocation

Phase	**Processing Task**	
	FPGA 1	**FPGA 2**
Acquisition	ISS Control	Accumulation
Processing	RTE Inversion	Pre-Processing

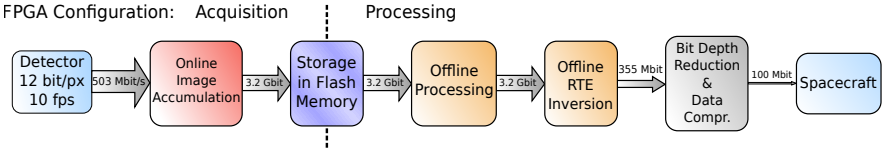


Figure 3.3.: Dataflow of Solar Orbiter *PHI* instrument

Out of these main tasks, the two tasks during processing phase, the inversion of the RTE and the Pre-Processing, are the most computing-intensive ones. While the inversion of the RTE is used to finally extract the scientific data while reducing the size of the data set, the pre-processing is used to prepare the accumulated data sets to give reliable results with high quality. Furthermore, the pre-processing is used for in-orbit calibration and re-focusing of the instrument's optics [61].

3.2.1. Data Acquisition

Albeit the data acquisition is not part of the regular pre-processing, it is important for the overall data flow of the instrument and will be shortly introduced in this section.
For applying the inversion of the RTE to extract the magnetic field vectors out of the sensor data, a set of multiple images is needed. Each of the images is recorded using different filter settings. Figure 3.4 shows an example of overall 24 images. The x-axis shows the four polarization states. The y-axis shows each of the polarization states filtered for 6 different wavelengths. For a better SNR, up to 24 images have to be accumulated within the hardware. This results in a total amount of $24 \times 24 = 576$ images captured from the sensor. Equation 3.1 depicts the acquisition process for the four polarization states and the different wavelengths λ_l with overall N iterations.

Accumulated images each have a size of up to 2048×2048 pixels and a 22 bit of depth. The data is captured by an Active Pixel Sensor (APS) which is connected by a *Channel-Link* interface. *Channel-Link* is a standard introduced by *National Semiconductor* which is used for differential, high-speed serial transmission. Sensor data is delivered with a depth of $12\ \mathrm{bit}$ at a clock frequency of $60\ \mathrm{MHz}$ at the input. The selected sensor and its readout electronics are presented in [62, 63].

$$I_{acc}(x, y, \lambda_l) = \sum_{i=0}^{N-1} \begin{pmatrix} I(x, y, \lambda_l, t_{4i}) \\ I(x, y, \lambda_l, t_{4i+1}) \\ I(x, y, \lambda_l, t_{4i+2}) \\ I(x, y, \lambda_l, t_{4i+3}) \end{pmatrix} \qquad \text{with} \qquad l = 0, \ldots, L-1 \tag{3.1}$$

Besides the simple accumulation, an optional filtering of Galactic Cosmic Rays (GCRs) has been implemented. It is assumed, that a sensor pixel which is hit by a GCR will only increase its intensity. Therefore, acquired pixel values are replaced by the mean value of the previously acquired pixel value if the difference to the mean vale exceeds a given threshold. In the special case of the second iteration, the pixel value is replaced by the minimum intensity of the first iteration pixel value and the new pixel value (with factor two). To minimize design complexity, the calculation of the mean value is implemented by using a simple look-up table.

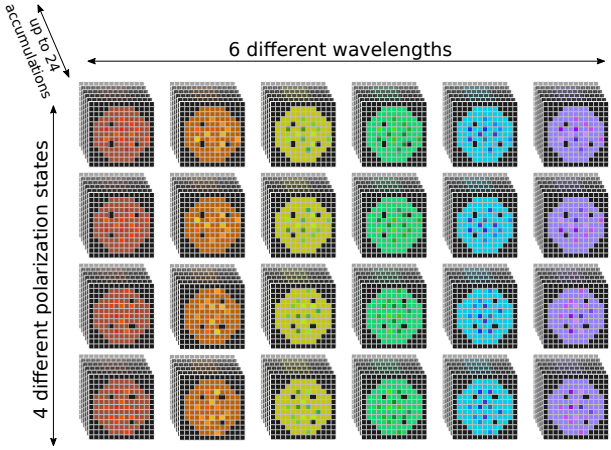
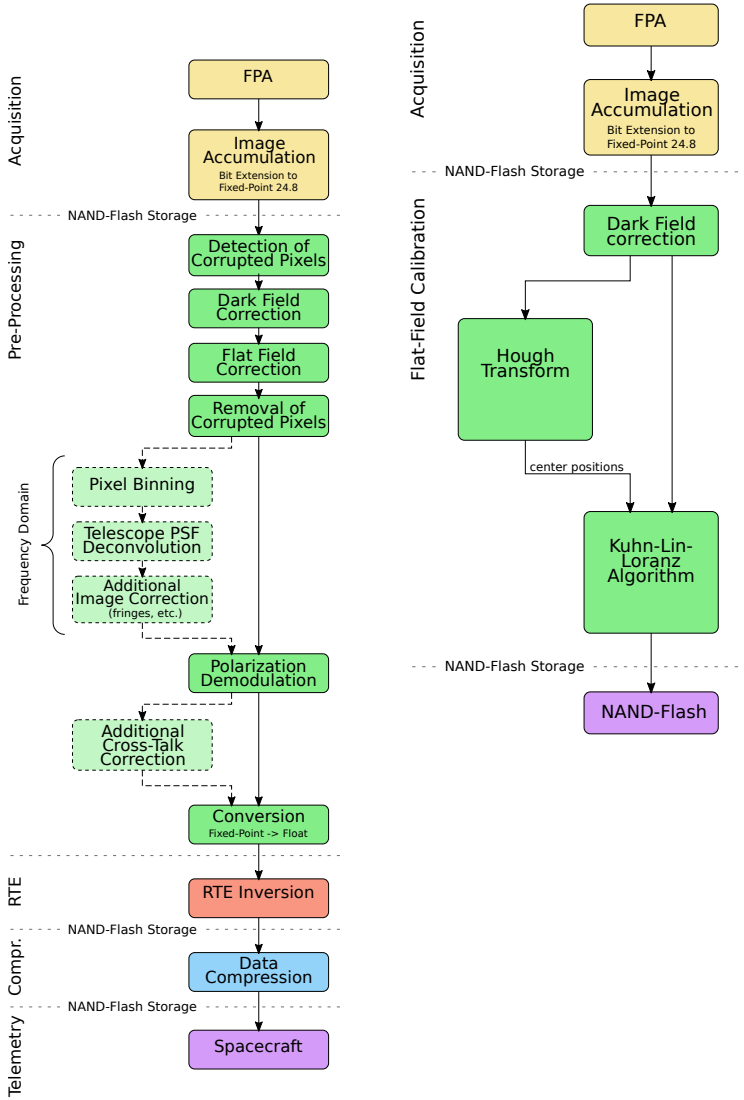


Figure 3.4.: Image accumulation during acquisition phase

3.2.2. Processing

After data is successfully acquired and stored to the NAND-Flash memory, computation of scientific data will be carried out during the processing phase. Therefore, the acquired data has to pass several processing steps, which might be different dependent on the current mode of operation. A typical flow for data processing is depicted in figure 3.5.



(a) Regular image processing [64] (b) Flat field calibration

Figure 3.5.: Image processing data flow

Dark and Flat Field Correction

The basic correction of raw sensor data is done through dark and flat field correction as shown in (3.2). The dark field $I_{dark}(x, y)$ consists of fixed-pattern noise caused by dark current and is independent of the wavelength and the polarization state. The pre-defined dark field will be subtracted from the accumulated image.
In contrast, the flat field $I_{flat}(x, y, \lambda)$ is caused by variations in the pixel-to-pixel sensitivity of the sensor and distortions of the optical path. The pre-defined flat fields are dependent of the spectral position λ and are stored as gain tables which are already corrected for dark field. The dark field corrected image has to be divided by the flat field.

$$I(x, y, \lambda) = \frac{I_{acc}(x, y, \lambda) - I_{dark}(x, y)}{I_{flat}(x, y, \lambda)} \tag{3.2}$$

Detection and Removal of Corrupted Pixels

Due to initial defects of the image sensor or other unforeseen effects, acquired image data may contain a few invalid pixel values. As these invalid pixels would have a drastic influence on the final results, the outliers have to be detected before dark and flat field correction and replaced before execution of calculations in the frequency domain. This has to be done by either application of an appropriate median filter or by interpolation of the neighboring pixels.

Pixel Binning

For reducing the resolution of the acquired data (e.g. for thumbnail generation), an optional pixel binning of images has to be available. To preserve scientific relevant information, the binning has to be carried out in the frequency domain.

Deconvolution of the Telescope PSF

The obtained images are affected by the optical system of the instrument. This corresponds to a convolution with the true image by the telescope Point Spread Function (PSF) as shown in (3.3).

$$I_{raw} = I_{true} \otimes \mathrm{PSF} \tag{3.3}$$

As this convolution can not be inverted by a simple division in the frequency domain, the deconvolution of the PSF is carried out using Wiener filters:

$$\begin{aligned} I^{\otimes}_{raw,PSF}(k,\lambda) &= I^{\otimes}_{raw}(k,\lambda) \cdot W(k) \qquad \text{with} \\ W(k) &= \frac{\mathrm{MTF}(k)}{\mathrm{MTF}(k)^2 + 1/\mathrm{SNR}} \end{aligned} \tag{3.4}$$

The Wiener filter will be calculated based on measurements performed on ground. It includes the Signal-to-Noise Ratio (SNR) as well as the Modulation Transfer Function (MTF)[1] which is defined by:

$$\mathrm{MTF} = |\mathrm{OTF}| = \left|\mathrm{PSF}^{\otimes}\right| \tag{3.5}$$

Additional Corrections

Due to unforeseen effects of the optics during the mission, e.g. fringing, additional corrections might be desired by scientists and applied through filtering in the frequency domain.

[1] (3.5) also includes the Optical Transfer Function (OTF) which is the Fourier transform of the PSF

Polarization Demodulation

The *PHI* instrument acquires four different polarization intensities. For further computations, these intensities $I_0 \dots I_3$ have to be converted into the four Stokes parameters I, Q, U and V. These parameters are acquired and calculated for each sensor position (x, y) and wavelength λ.
The conversion is carried out by multiplication of the measured intensities with a demodulation matrix in dependency of the position (x, y) and the wavelength, as shown in (3.6). The demodulation matrix is dependent on the optical system which might alter due to different ambient temperatures and degradation.

$$I_S(x,y,\lambda) = \begin{pmatrix} I(x,y,\lambda) \\ Q(x,y,\lambda) \\ U(x,y,\lambda) \\ V(x,y,\lambda) \end{pmatrix} = D(x,y) I_{raw}(x,y,\lambda)$$

$$= \begin{pmatrix} D_{0,0}(x,y) & D_{0,1}(x,y) & D_{0,2}(x,y) & D_{0,3}(x,y) \\ D_{1,0}(x,y) & D_{1,1}(x,y) & D_{1,2}(x,y) & D_{1,3}(x,y) \\ D_{2,0}(x,y) & D_{2,1}(x,y) & D_{2,2}(x,y) & D_{2,3}(x,y) \\ D_{3,0}(x,y) & D_{3,1}(x,y) & D_{3,2}(x,y) & D_{3,3}(x,y) \end{pmatrix} \cdot \begin{pmatrix} I_0(x,y,\lambda) \\ I_1(x,y,\lambda) \\ I_2(x,y,\lambda) \\ I_3(x,y,\lambda) \end{pmatrix} \tag{3.6}$$

Additional Crosstalk Correction

Even though instrumental effects are taken into account by the demodulation matrix, the calculated Stokes parameters might include residual crosstalk error of unknown origin. For further computation, only the crosstalk from Stokes V to Stokes Q and U is relevant, as shown in (3.7).

$$Q = Q_{true} + aV \qquad \text{and} \qquad U = U_{true} + bV \tag{3.7}$$

The constants a and b have to be estimated, which can be done by linearly fitting Q and U to V. Therefore, the minima have to be calculated as in (3.8).

$$\sum_{x,y}(Q - aV)^2 \qquad \text{and} \qquad \sum_{x,y}(U - bV)^2 \tag{3.8}$$

This is finally carried out by partial differentiation and solving the resulting system of linear functions. Thus, a and b are calculated as in (3.9).

$$b = \frac{n\sum_{x=0}^{n-1} Q(x)V(x) + \sum_{x=0}^{n-1} Q(x) \sum_{x=0}^{n-1} V(x)}{n\sum_{x=0}^{n-1} Q(x)^2 - (\sum_{x=0}^{n-1} Q(x))^2}$$

$$a = \frac{1}{n}(\sum_{x=0}^{n-1} V(x) - b\sum_{x=0}^{n-1} Q(x)) \tag{3.9}$$

Inversion of the RTE

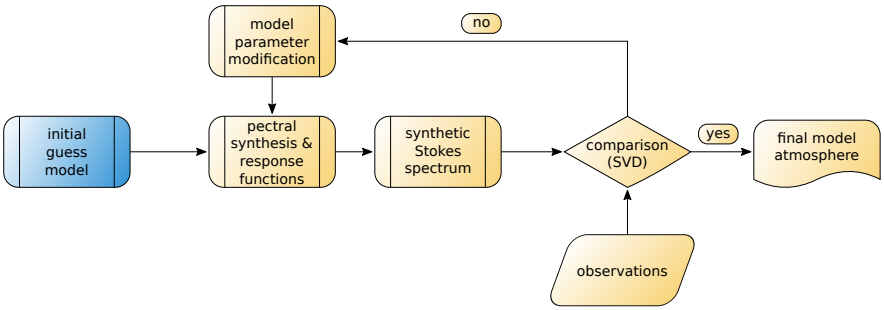


Figure 3.6.: Basic principle of the RTE inversion according to [65]

The inversion of the RTE used for the *PHI* instrument is an iterative approach based on the Milne-Eddington solution of the RTE [66, 67]. The inversion is implemented by *IAA* as a dedicated FPGA design based on the software implementation *C-MILOS* which is presented in [68]. As the iteration is carried out independently for each pixel of the 24 input images, the algorithm has no need for buffering of complete images. Thus, the implemented SIMD architecture follows a different approach based on several parallel processors (so called *nProcessors*) [65] which is summarized in section 3.4.4.
The basic principle of the iterative approach is shown in figure 3.6. It is based on an initial guess model which is modified until the observations match the spectrum derived from the model.

3.2.3. On-Board Instrument Calibration

Because calibration data might be susceptible to ambient temperature and degradation effects (of e.g. the optics), it has to be acquired autonomously in-flight [69]. The calibration data set consists of dark and flat field data which has to be acquired separately for FDT and HRT. Especially the determination of the flat field turns out to be difficult, as this is usually carried out with the help of a uniformly illuminated surface. As this is not available during the mission, other techniques such as the flat-field algorithm according to Kuhn-Lin-Loranz [70] have to be used instead.
Besides the correction of the acquired data, also the instrument's focus has to be set up before each data acquisition, which requires edge detection, determination of the center of the solar disk, as well as calculation of the gradient.

3.3. Requirements

Excluding the data acquisition and the inversion of the RTE, which are dedicated FPGA designs, the previously described processing steps can be summarized to the mathematical functions as listed in table 3.2. More detailed information about the operations needed for *PHI* can be found in [64].
Due to the telemetry rate and downlink capacity allocated to the instrument, the regular processing aims at a rate of about processing one complete dataset (24 images á 2048×2048 pixels) every 15 minutes. Nevertheless, scientific goals could profit from higher processing capabilities. Besides the regular processing, in-orbit instrument calibration will only be performed at certain times and thus, processing time is less critical.
The desirable processing framework also needs to implement the few mathematical complex functions from table 3.2 which are needed during instrument calibration. This includes determination of the center position of the solar disk, as well as calculating the flat field with the Kuhn-Lin-Loranz Algorithm (KLL). Besides the listed mathematical functions, also additional conversion to feed the dedicated RTE inverter and compression designs have to be considered. This includes conversion between fixed and floating pont format, as well as reordering of image pixels. Because the processing can only be tested on ground with pre-estimated input data generated from other telescopes, the processing also has to be prepared to respond to unforeseen effects. Therefore, a high grade of adaptability throughout the mission is needed.
As the summarized functions will hardly fit into one FPGA configuration of the used *Virtex-4 SX55*, the desired framework is required to dynamically exchange its functionality. As the transfer of data between the processing FPGA and the NAND-Flash memory board is relatively slow, this has to be done without storage of interim results in the NAND-Flash memory.

Table 3.2.: Overview of mathematics needed for image pre-processing

Description	**Mathematical Operation**
Adding entire images	$I_o(x,y) = I_a(x,y) + I_b(x,y)$
Subtracting entire images	$I_o(x,y) = I_a(x,y) - I_b(x,y)$
Division of entire images	$I_o(x,y) = I_a(x,y) \,/\, I_b(x,y)$
Multiplication (complex) of images	$I_o(x,y) = I_a(x,y) \cdot I_b(x,y)$
Scalar addition of images	$I_o(x,y) = I_a(x,y) + c$
Scalar subtraction of images	$I_o(x,y) = I_a(x,y) - c$
Scalar multiplication of images	$I_o(x,y) = I_a(x,y) \cdot c$
Scalar division of image pixels	$I_o(x,y) = I_a(x,y) \,/\, c$
Intensity thresholding	$I_o(x,y) = \begin{cases} 1 & : I_{th1} \leq I_a(x,y) \leq I_{th2} \\ 0 & : \text{else} \end{cases}$
Fast Fourier Transform of images	$I^{\otimes}(k_x,k_y) = \mathscr{F}\{I(x,y)\}$
Median filtering of images	$I'(x,y) = \text{Median}\{I(x,y)\}$
4×4 Matrix Operations	$\begin{pmatrix} I \\ Q \\ U \\ V \end{pmatrix} = \begin{pmatrix} D_{0,0} & D_{0,1} & D_{0,2} & D_{0,3} \\ D_{1,0} & D_{1,1} & D_{1,2} & D_{1,3} \\ D_{2,0} & D_{2,1} & D_{2,2} & D_{2,3} \\ D_{3,0} & D_{3,1} & D_{3,2} & D_{3,3} \end{pmatrix} \cdot \begin{pmatrix} I_0 \\ I_1 \\ I_2 \\ I_3 \end{pmatrix}$
Cropping of images	$I_o(x,y) = I_i(x,y) \quad \forall\, x \in [x_l, x_r] \,\wedge\, y \in [y_t, y_b]$
Least squares fitting	$\chi^2 = \min_a \sum_{x,y} (q - aV)^2$

Table 3.2 – continued

Minimum determination	$\min(I(x,y))$
Maximum determination	$\max(I(x,y))$
Mean intensity of images	$\langle I \rangle = \frac{1}{N} \sum_{x,y} I(x,y)$
Gradient of images	$\nabla I(x,y) = \left(\frac{\partial I(x,y)}{\partial x}, \frac{\partial I(x,y)}{\partial y} \right)$
Logarithm	$I_o(x,y) = \log(I_i(x,y))$
Square-root	$I_o(x,y) = \sqrt{I_i(x,y)}$
Standard deviation of images	$\sigma = \sqrt{\frac{1}{N-1} \sum_{x,y} (I(x,y) - \langle I \rangle)^2}$
Kuhn-Lin-Loranz constant	$G^{r+1}(x) = K(x) + \frac{1}{n(x)} \sum_{i<j} [G^r(x - a_i + a_j) + G^r(x - a_j + a_i)]$
Kuhn-Lin-Loranz iteration	$K(x) = \frac{1}{n(x)} \sum_{i<j} [[D_i(x) - D_j(x - a_i + a_j)] + [D_j(x) - D_i(x - a_j + a_i)]]$
Hough transform	—
Histogram	—
Thumbnail generation	—

3.4. Framework Architecture

For the given constraints of the *PHI* DPM, a suitable data processing architecture has to be developed and evaluated in consideration of the specified requirements given in section 3.3. The previous defined requirements of the image processing pipeline lead to a simple architecture, containing an SDRAM memory controller, the defined set of operations and an interface to the NoC. The principal architecture of the processing pipeline is depicted in figure 3.7 and was presented in [71].
All required image data and processing parameters are buffered in the image memory buffer SDRAM with a capacity of $8\,\mathrm{Gibit}$, directly connected to the pre-processing RFPGA. For fast access the memory is addressed in words of $64\,\mathrm{bit}$ and includes a single-symbol Reed-Solomon error correction. Since different configurations of the FPGA may need to be loaded during data processing but data in the buffer memory should be hold, the memory controller is able to switch the SDRAM devices into self-refresh mode during reconfiguration of the RFPGA. In order to allow a parallel operation of the configured processing modules, the buffer memory has to be accessed simultaneously from different read- and write-ports. Thus, a multi-port memory interface is implemented, which has a configurable number of read- and write-ports, served by simple round-robin scheduling scheme. Typically, the processing modules need to be implemented with two read- and one write-interface (e.g. pixel-wise addition), each including address and data. Other options are configurations with one read- and one write-port (e.g. FFT/IFFT) or only one port for reading of image data (e.g. find minimum/maximum pixel value).
All included operation modules are connected to the main *GR712* CPU (and thus, the software) by a *SoCWire* switch. While the communication with the processing modules is restricted to simple reading and writing of control and status registers, the transfer of larger amount of data between the central NAND-Flash mass memory and the local RFPGA memory buffer is handled by a separate *SoCWire* module.

3.4.1. Integration into the Subsystem

The modular architecture presented in the previous section is integrated into the overall system to enable a reliable and easy to use integration into the overall DPM subsystem and especially a seamless interaction with the on-board software. Therefore, the overall framework not only consists of several FPGA configuration bitstreams each containing a set of processing modules. Furthermore, it also consists of a software Application Programming Interface (API) handling i.a. the module status and control via the involved *SoCWire* registers. For data transfer, all the pre-processing FPGA designs are equipped with an identical module for buffer memory read/write operations including calculation of a Cyclic Redundancy Check (CRC) checksum for verification. This module is always located at *SoCWire* hardware ID `0xD6` and also holds general information such as number of included processing modules and *Subversion* (*SVN*) revision number. All other processing modules are logically placed in arbitrary order at *SoCWire* hardware ID `0x80` and upwards. In addition, each module holds a unique identifier in *SoCWire* register `0x1c`. This enables the software API to automatically detect the included processing modules of a pre-processing bitstream after successful RFPGA configuration. Therefore, execution of modules which are non-existent in the currently loaded RFPGA configuration will be handled as an exception by the software API.
Specific modules are executed by a call to the corresponding function of the API. The software will then keep control of the sequential execution by setting the related registers (i.e. read/write addresses, length, mode of operation) in dependence of the given parameters. The API is blocking until the execution is finished which is signaled by an interrupt to the CPU. After read-out of status information (i.e. overflow) and optional scalar results, the API function will return with the corresponding status. Additionally, an automatic self-test may be executed automatically to ensure proper operation of each module (described in more detail in section 3.7). Generally, also a parallel execution of the modules would be possible. Because dependencies in the processing chain would increase the complexity of software control while only gaining little performance (due to limited data rate at the memory interface), this is not used within the *PHI* DPM.

Each specific pre-processing FPGA design which contains a defined set of processing modules has to be synthesized manually by using the *Xilinx ISE* tool-chain. All the needed resources are included in a global library which contains VHDL descriptions and IP-cores of various system components (i.e. SDRAM memory controller, *SoCWire* switch and protocol handler) and the actual processing modules. The *Makefile* based tool-flow for synthesis and simulation is derived from the *GRLIB* which is available from *Gaisler*. This tool-flow has been extended by a *Python* script for the automatic generation of the pre-processing design entity. This allows the fail-safe composition and setup of the overall *ISE* project for each RFPGA bitstream by simple parametrization in the *Makefile*.

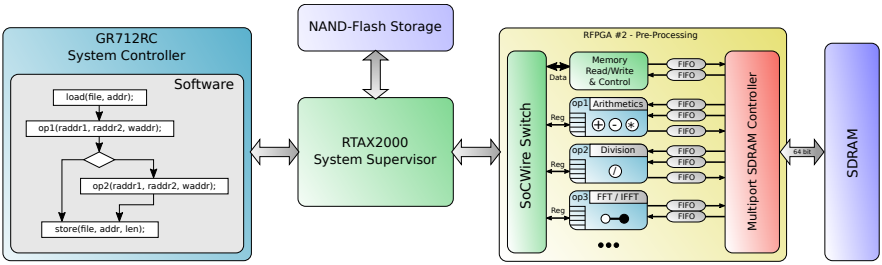


Figure 3.7.: Architecture of the Image Pre-Processing

3.4.2. SDRAM Buffer Memory and Controller

For fast image processing, an SDRAM buffer memory with a total gross capacity of 12 Gibit is attached to the pre-processing RFPGA. The memory is organized in four SDRAM stacks as shown in figure 3.8. Internally, each of the four *3D-Plus* memory stacks contains six commercial *ISSI IS42S16320B* memory devices [72]. The internal structure of a single stack is shown in figure 3.9. Generally, these four attached memory stacks can be used in two ways:

- Concurrently as two independent, 48 bit wide memories with 6 Gibit capacity each
- Combined as one single, 96 bit wide memory with 12 Gibit capacity

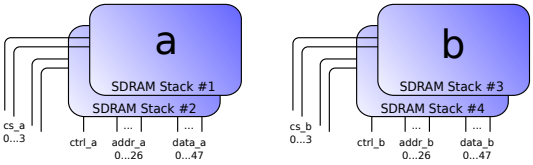


Figure 3.8.: Organization of SDRAM stacks

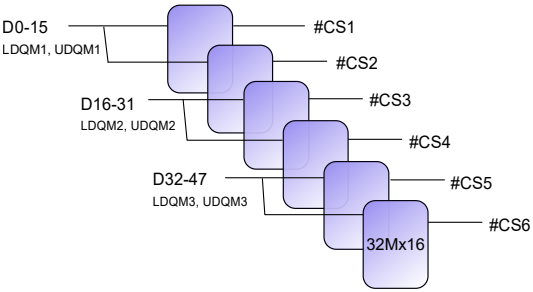


Figure 3.9.: Organization of a single SDRAM stack

For a simple and effective design, the pre-processing makes use of the memory stacks as a single 96 bit wide memory. For mitigation of SEUs, the contents will be protected by two independent Reed-Solomon Error-Correcting Code, consuming 16 bit each. This results in a storage to two 32 bit image pixels for real image data, or one 64 bit complex image pixel per address. Therefore, the net capacity of the buffer memory is 8 Gibit. By using two independent error corrections, one of the two 32 bit words can be masked during a write access to the memory.

For operation of the attached memory devices, a specialized SDRAM memory controller together with a multi-port interface has been implemented. In order to fulfill the required processing performance while providing reliable operation, the SDRAM controller is clocked at $50\ \mathrm{MHz}$ and supports the following features:

- Individual addressing
- Seamless, overlapping read/write access
- Scrubbing of the internal mode register
- Support for self-refresh and power-down
- Masking of lower/upper word
- Independent Reed-Solomon EDAC for lower/upper word
- Optional row activation of all chips

To allow memory access from all modules included in the FPGA configuration, the multi-port interface takes care of proper scheduling. This is implemented in a simple Round-Robin manner with a configurable burst count. The number of read and write ports can be configured by generics which allows a simple adaption of FPGA designs including different modules. Both, the multi-port interface with the associated FIFOs and the controller logic are shown in the block diagram of figure 3.10. The same SDRAM controller is also used in the design of the data acquisition, albeit with a simplified multi-port interface.

While reading and writing data from and to the memory, care should be taken of the row activation time. A single row has the size of $2^{10} = 1024$ words. Therefore, a single line of a real, 2048 real image pixels will fit into one row. When accessing image data out-of-order (e.g. vertical pixel access instead of horizontal), row activation penalty (3 cycles per access) has to be considered, resulting in a reduced throughput. This can be optimized by optionally enabling the SDRAM controller to activate the current row in all attached SDRAM chips at the same time. This increases the effective size of the row by a factor of four (2048×4 pixels per row) but also increases the power consumption.

To assure the SDRAM will keep its contents while performing a reconfiguration of the *Virtex-4* FPGA, the devices can be switched to self-refresh mode before reconfiguration.

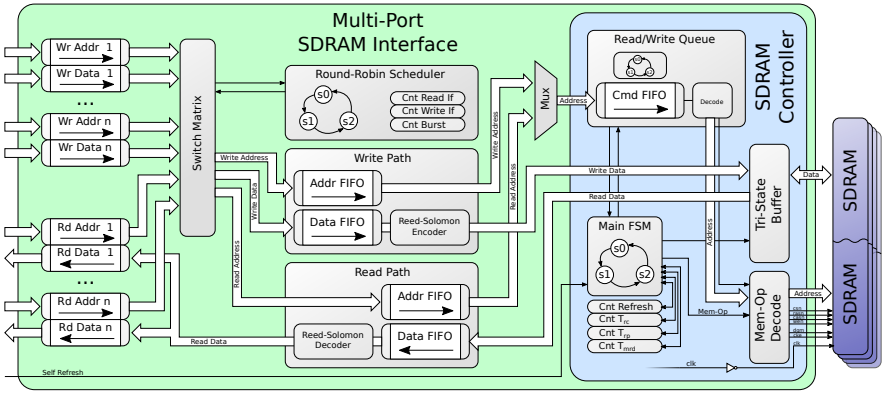
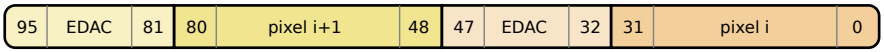


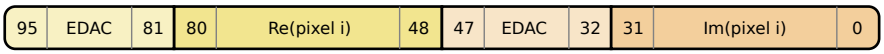
Figure 3.10.: Block diagram of multi-port SDRAM interface and controller

Image Pixel Format

Figure 3.11a shows the composition of real image data inside the buffer memory. Invalid pixels (overflow, underflow) are marked by setting the value to `0x80000000`.
The composition of complex image data within the buffer memory is depicted in figure 3.11b. Invalid pixel components (overflow, underflow) are marked by setting the value to `0x80000000`, which is defined as NaN (Not a Number).



(a) Pixel format of real image including EDAC



(b) Pixel format of complex image including EDAC

Figure 3.11.: Image pixel format

3.4.3. Tool-Flow

Besides the technical implementation of the components included in the processing framework, this section covers the basic tool-flow which is used to generate, simulate, test and evaluate FPGA bitstreams. The basic concept of this tool-flow is depicted in figure 3.12.
Principally, the project files for synthesis and place & route (*Xilinx ISE*) and RTL simulation (*ModelSim*) are generated by a set of Makefiles and scripts within the structure provided by the *GRLIB* from *Cobham Gaisler* [73]. This library and its tool-flow is mainly used to allow co-simulation with a *LEON3* CPU which is also used within the project. The *GRLIB* has been extended by a simple *Python* script which generates needed glue-logic in form of a `.vhdl` file in dependance of a list of up to 16 modules specified in a *Makefile*. In this way, multiple, individual configuration bitstreams can be generated inside separate project folders. The processing modules which are desired to be included into the corresponding bitstream are altogether located inside the *GRLIB* folder structure. The module interface is standardized but can contain a variable count of read and write interfaces to the multi-port SDRAM memory controller which, on his part, is also parametrizable.
The generated designs can be simulated using *ModelSim*. For convenience, module specific test stimuli can be generated by a set of external *Python* scripts. After successful simulation, configuration bitstreams can be generated by running synthesis and place & route from within *Xilinx ISE*. The resulting FPGA configurations can finally be tested and evaluated using a *Python*-based test-framework running on ground support environment which will be covered in section 3.8.1.

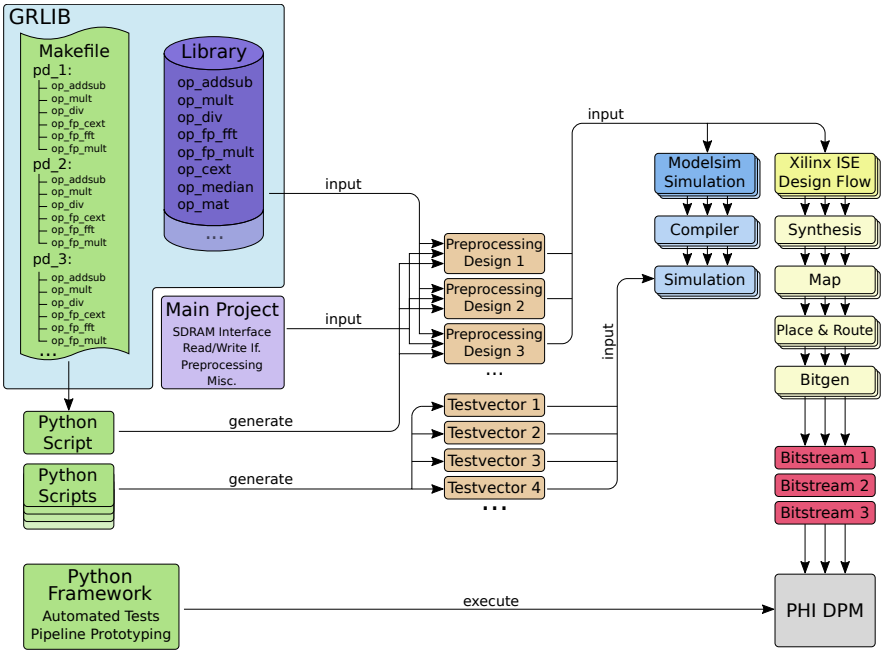


Figure 3.12.: Tool-flow for generation of different processing designs

3.4.4. Alternative Implementation Variants

Because the dedicated implementation of the individual processing modules in RTL is quite extensive, this section shall illustrate two possible alternatives.

GPP Architecture

One alternative to the dedicated implementation of modules could be the integration of a general-purpose processor into the processing architecture. A possible candidate is the commonly used *LEON2/3* IP-core from *Gaisler* which has been evaluated to run on one of the FPGAs in [74]. Furthermore, the proprietary *Xilinx MicroBlaze* architecture or a lightweight implementation of the relatively new *RISC-V* ISA are other possible options.

Usually, the implementation of a processor core within an FPGA has less performance than the realization as a dedicated ASIC. Also, many soft-processors aiming for FPGA integration are 32 bit architectures which would not necessarily benefit from the given 64 bit memory interface. However, the integration of a light-weight processor core into the presented framework for the computation of less timing-critical functions could be a possible extension.

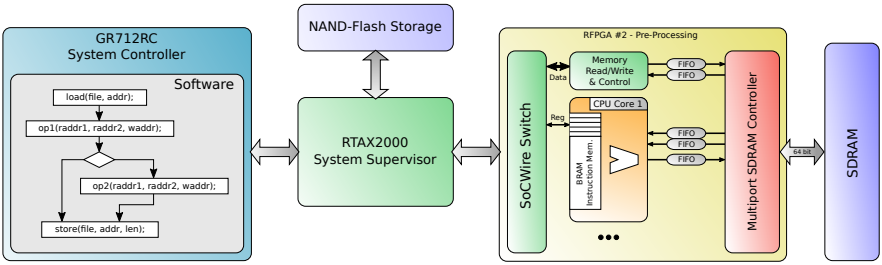


Figure 3.13.: Alternative architecture using general purpose CPU cores

Specialized SIMD Nano-Processors

For carrying out the time critical inversion of the Radiative Transfer Equation (RTE) which is part of the processing on-board the *PHI* instrument, an approach using a SIMD multiprocessor architecture was used by *IAA* [75].
The proprietary architecture is using overall 12 nano processing cores inside the *Virtex-4 SX55* FPGA of the data processing module on-board the *PHI* instrument. These processors make use of a common instruction ROM to implement the actual algorithm for the inversion of the RTE. Each processor core contains its own memory, leading to a distributed memory SIMD architecture. Transfer of the scientific data to and from other DPU components is done by a *SoCWire* interface. A set of special functions is included in a separate block and are shared between the processors. The general architecture is shown in figure 3.14.
While proposed nano-processors include a simple floating-point Arithmetic Logic Unit (ALU) (single-precision addition, subtraction and multiplication),

the shared operations block includes division, arctan, sine, cosine, square root and a whole Singular Vector Decomposition (SVD) algorithm of a covariance matrix. While division and SVD are carried out in floating-point, the other components are implemented in fixed-point and therefore need a conversion from floating-point values and vice versa.
Each of the twelve processor cores are controlled by a dedicated enable signal which is encoded into the instruction word. The architecture has separated instruction, data input and data output buses. Access to the data buses is controlled by the processor enable signals. Thus, the access to the shared operations block is restricted to one processor at a time. In the special case of the RTE inversion, no data needs to be exchanged between the processing cores. Therefore a communication between the cores is not foreseen but could be implemented by a bridge between the input and output data bus. Only one transfer between two processors per clock cycle would be possible and thus, the performance of the architecture is strongly dependent on the the user's algorithm.

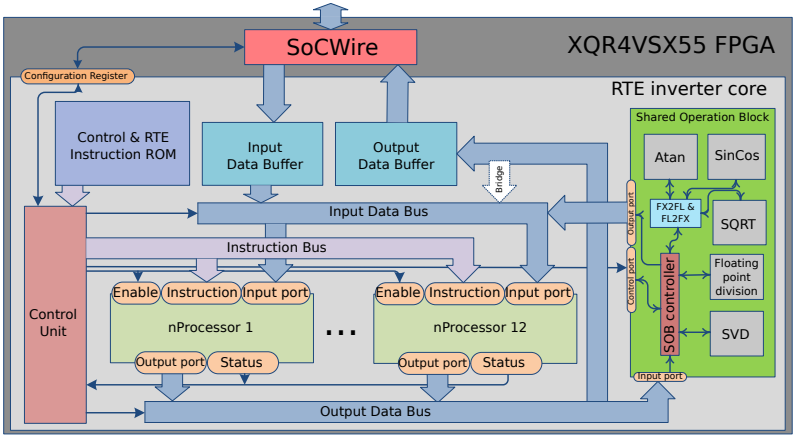
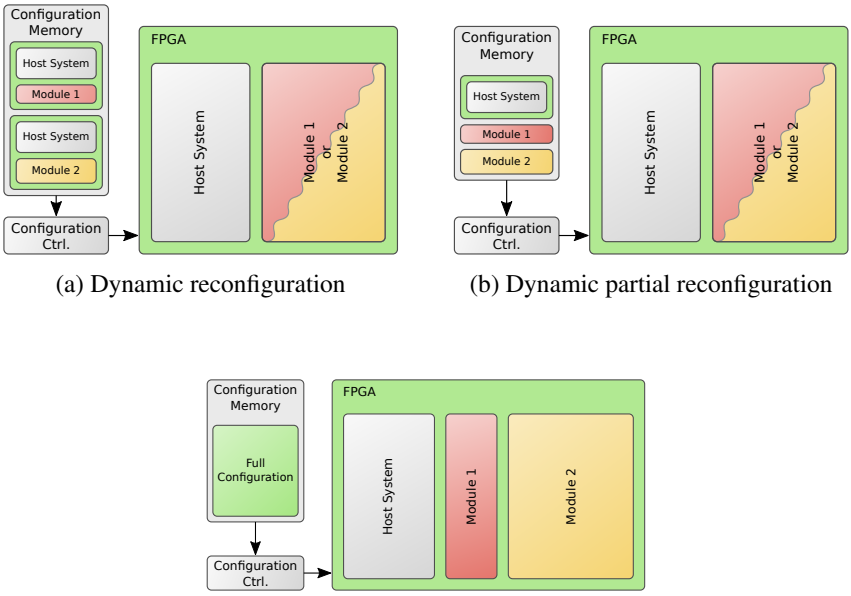


Figure 3.14.: SIMD architecture used for RTE inversion [75]

3.5. Dynamic Reconfiguration

Complex systems typically consists of multiple functional modules which are connected by a dedicated communication architecture [50]. Today's modern FPGA platforms are highly integrated and provide a large number of logic slices as well as integrated DSP and RAM blocks. Therefore, such a complex design might fit into a single device. In many applications not all of the modules need to operate concurrently. In this case, an unused module in the design wastes logic resources, power and therefore costs.
At this point, the great advantage of SRAM-based FPGAs is the ability for dynamic partial reconfiguration. This allows to exchange specific modules in the FPGA while the remaining design is still in operation.
Various configuration concepts are depicted in figure 3.15. A full static FPGA configuration without the use of reconfiguration is shown in figure 3.15c. In this case, all the needed modules are included in one FPGA design with the cost of utilizing a higher amount of resources. The FPGA configuration is stored inside an external configuration memory and loaded on start-up. In contrast, figure 3.15b shows the use of dynamic partial reconfiguration. A host design with the static design components are loaded from the configuration memory once during start-up. Afterward, specific modules can be loaded at run-time when needed. The area of the reconfigurable FPGA has to be divided into several rectangular regions of fixed size. The minimum size is defined by the largest module to be loaded into the specific area. Each region already has to contain the required amount of RAM and DSP blocks needed for the modules to be fit into this reconfigurable slot. Furthermore, each reprogrammable area also has to be connected to the host system by a predefined interface which is isolating the module by the instantiation of specific bus macros. Consequentially, dynamic partial reconfiguration requires an increased design effort as well as a more complex configuration management. It also demands for careful module testing, as there are several combinations of processing modules, which might have an influence to each other.
Even though today's FPGAs provide a high gate count, there might not be enough resources for an entire system to be implemented in a single device. This is also the case for the introduced architecture of a flexible processing pipeline, which consists of a high quantity of different processing modules.

In the case of the processing pipeline it is foreseen to use only one processing module at a time. The main reason is limited data-rate to and from the SDRAM buffer memory. Besides this, it also keeps the handling of the processing steps sequential and therefore reduces the risk of unsatisfied dependencies. Because only one processing step is active at a time, the resource usage would only be little efficient. Thus, the use of dynamic in-flight reconfiguration will not only improve the resource utilization of the processing pipeline, it generally enables to use all the required modules in a time-shared approach.



(a) Dynamic reconfiguration

(b) Dynamic partial reconfiguration

(c) Full configuration / no reconfiguration

Figure 3.15.: Various methods of (re)configuration

As discussed in section 2.6.5, the *PHI* DPM anyway uses dynamic partial reconfiguration for scrubbing the configuration memory contents of the used *Virtex-4* FPGA. The switching of the two RFPGAs between the different modes of instrument operation, data acquisition and processing, is usually done within an interval in the order of several days. Therefore, data will be safely stored in NAND-Flash memory of the instrument during reconfiguration and RFPGAs might also be powered off for a specific time in between. For this reason, a full reconfiguration of the devices is fully adequate.
This is different when switching between multiple FPGA configurations of the presented processing framework. A fast partial reconfiguration of the RFPGA executing the processing modules could be of interest. The implemented scrubbing and configuration controller is generally able to also perform partial reconfiguration of the two RFPGAs. The overhead induced through the full reconfiguration of the pre-processing FPGA in the case of the processing on-board the *PHI* instrument is negligible compared to the overall processing time, which will be discussed in section 5.1. Therefore, a full reconfiguration of the FPGA with a set of precompiled configuration bitstreams is fully sufficient.
The important prerequisite for a full reconfiguration during active pre-processing is that the contents of the attached volatile SDRAM buffer memory have to be kept safely in storage when the FPGA is being reconfigured. While throughout a partial reconfiguration the responsible DRAM memory controller still operates and will generate the mandatory refresh cycles for the dynamic memory to keep its contents, these refresh cycles will be interrupted during a full reconfiguration of the FPGA. While the full reconfiguration time usually takes around $t_{conf,max} = 3\text{s}$, the refresh has to be triggered continuously after a maximum of $t_{ref} = 64\text{ms}$ [1].
A solution to overcome this gap is to activate the built-in self-refresh and power-down mode which can be found on most SDRAM memory devices. When switched into the self-refresh mode, special control lines (e.g. chip-select) have to be kept at logical high level to avoid the memory devices from returning into standard operation. This can be done by using external pull-up resistors on the needed control lines. The *Virtex-4 SX55* FPGA, as used for the *PHI* DPU, can be externally configured for safe hot-swapping. In this case all IO pins of the device will activate an embedded weak pull-down resistor.

[1] Valid for the used *ISSI IS42S16320B* SDRAM memory component [72]

3.6. Implementation of Image Processing Modules

To cover the full functionality of data processing, especially for image processing, the medium-level processing steps have to be divided into a set of basic functions. On the example of the *Solar Orbiter PHI* instrument, this results in a large number of dedicated modules, which have to be implemented in dedicated logic hardware onto the FPGA. An excerpt of dedicated modules and their corresponding functions (function name as called within C-code) needed for data processing on the *PHI* instrument is depicted in table 3.3.
This section will give a detailed insight into the implementation of the processing modules. Therefore, a few examples from processing of the *PHI* instrument will be presented.

Table 3.3.: Excerpt of processing modules needed for the *PHI* instrument

Module/Function Name	Description
op_addsub	
add()	addition of images
add_sc()	addition of scalar value to images
sub()	subtraction of images
sub_sc()	subtraction of scalar value from images
op_mult	
mult()	multiplication of images
mult_sc()	multiplication of images by scalar value
cmult()	complex multiplication of images
cmult_sc()	complex multiplication of images by scalar value
complex_conj()	complex conjugate of pixels
op_fp_mult	
fp_mult()	floating-point mult. of images
fp_mult_sc()	floating-point mult. of images by scalar value
fp_cmult()	cplx. floating-point mult. of images
fp_cmult_sc()	cplx. floating-point mult. of images by scalar value
fp_complex_conj()	cplx. floating-point conjugate of pixels
op_div	
div()	division of images

Table 3.3 – continued

`div_sc()`	division of images by scalar value
op_thold	
`thold()`	threshold (<,>,=) against scalar value
`thold_rpl()`	threshold (<,>,=) with replacement
op_fpcext	
`real2fpcplx()`	extension of real values to complex pixel format
`fpcplx2real()`	reduction of complex values to real pixel format
op_fp_fft	
`fp_fft()`	1D Fast Fourier Transform (fl.-point if.)
`fp_ifft()`	inverse 1D Fast Fourier Transform (fl-point if.)
op_sqrt	
`sqrt()`	square root of image pixels
op_log	
`log()`	logarithm of image pixels
op_fix2float	
`fix2float()`	conversion to floating-point and reordering
op_crop	
`crop()`	cropping of images
op_multisum	
`min()`	minimum value and position of image
`max()`	maximum value and position of image
`sum()`	sum of image pixel values
`mean()`	mean of image pixel values
`lsfit()`	linear least squares fitting
`rms()`	root mean square
`sd()`	standard deviation
op_logic	
`logic()`	logical (`AND`, `OR`, `XOR`, `NAND`, `NOR`, `XNOR`)
`logic_sc()`	scalar logical functions
op_shift	
`shift()`	hor./vert. shift of image
op_median	
`median_3x3()`	3×3 median filtering of images
`median_5x5()`	5×5 median filtering of images
op_binning	
`binning()`	simple 4:1 pixel binning of images
op_histogram	
`histogram()`	histogram with 64 bins
op_tmr	
`tmr()`	TMR voting of images

3.6.1. Basic Arithmetic Operations

Basic processing of the acquired image data can be done with general, pixel-wise arithmetic fixed-point operations. These operations include addition, subtraction, multiplication and division of image pixels.
Because these kind of operations can be implemented inside an FPGA without high effort, this kind of module is well suited to explain the basic architecture of a processing module. This shall be done using the example of pixel-wise and scalar addition/subtraction of images. The principal design of the addition/subtraction module is depicted in figure 3.16.
Because this operation has two inputs and one output, it is referred to as a binary operation (not to be confused with bitwise operation). Therefore, the corresponding module is connected to two input channels and one output channel. Each channel is represented by a pair of FIFOs, one for carrying address information and the other for carrying the actual data.
Before execution of the module, the module has to be parametrized via its *SoCWire* interface. This typically includes start addresses for reading and writing, the length of the image data vector and the operation mode (e.g. addition, subtraction, scalar mode). When the module is set up, execution is triggered by setting a dedicated *SoCWire* register. In case of the addition/subtraction module, overall three address generators will be set up with its corresponding start address and length and start address generation.

As soon as read-out pixel data arrives at the two data input FIFOs, result will be calculated for two image pixels in parallel using two *DSP48* blocks from the *Virtex-4* fabric. If a defined NaN value is detected at the input or an overflow is detected, the output is set to the defined NaN value. Additionally, a status bit will be set for the software function to return with overflow status. When in scalar mode, only one read address generator will be activated on execution and the DSP blocks will be fed with a scalar value instead. When overall image length is finally reached, address generation will be stopped and an interrupt will be triggered to inform the software layer that execution has finished.

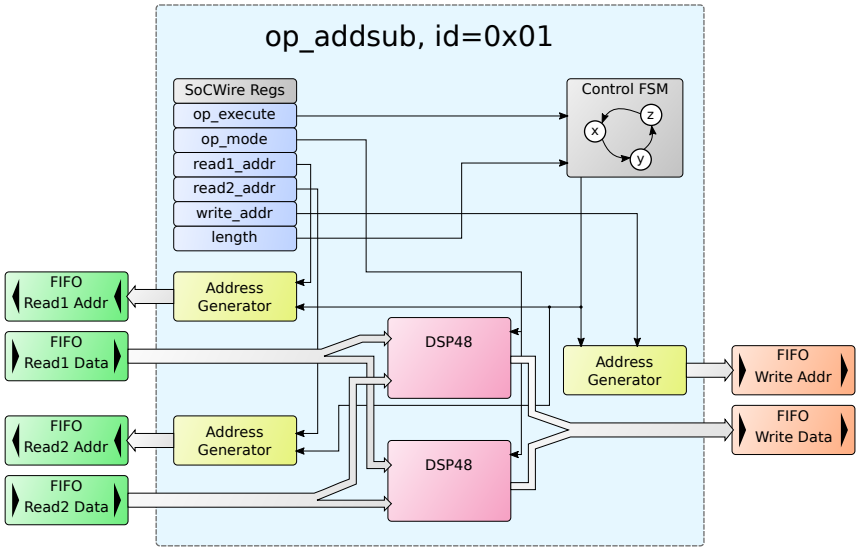


Figure 3.16.: Insight into module for addition and subtraction of images

3.6.2. Multipurpose Sum Module

Summing up the overall pixel values of an input data set is often needed during processing for image statistics and linear fitting of data sets. Also finding the minimum/maximum pixel value of an image and its position might be needed in several steps for the parameterization of further processing steps. All these operations have in common that no resulting vector data is written back to the buffer memory. Accumulated scalar values are instead buffered locally inside the module. Therefore, it is apparent that these operations can be combined into one single module. Because this module only has read channel inputs and the results are directly returned to the software layer, this category of modules is slightly different than the example previously given in section 3.6.1. On the example of the *PHI* instrument, this module covers the calculations listed in table 3.4.

Table 3.4.: Calculations covered by multi purpose sum module

Minimum / Maximum	$\min(x), \quad \max(x), \quad \mathrm{pos}_{\min}(x), \quad \mathrm{pos}_{\max}(x)$
Sum / Mean Value	$\mathrm{sum} = \sum x_i, \qquad \mu = \frac{1}{N}\sum x_i$
Root Mean Square	$\mathrm{rms} = \sqrt{\frac{\sum {x_i}^2}{N}}$
Standard Deviation	$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i-\mu)^2} = \sqrt{\frac{\sum_{i=1}^{N} x_i^2}{N} - \mu^2}$
Linear Least Squares Fitting	$b = \frac{N\cdot\sum x_i\cdot y_i - \sum x_i \cdot \sum y_i}{N\cdot\sum {x_i}^2 - (\sum x_i)^2}, \qquad c = \frac{\sum y_i - b\cdot\sum x_i}{N}$

All the listed quantities can be calculated by software, where only the following parameters have to be determined in the hardware module for acceleration:

- $\min(x)$, $\max(x)$ and its positions $\mathrm{pos}_{\min}(x)$, $\mathrm{pos}_{\max}(x)$
- Each sum $\sum x_i$, $\sum y_i$, $\sum {x_i}^2$ and $\sum (x_i \cdot y_i)$
- The total count of valid pixels N

Pixels marked as NaN shall not be included in the parameter determination, therefore N is not necessarily equal to the overall length of the data set and has to be counted separately. Because the calculation of the linear Least Squares Fitting requires an additional input y, this second input has to be switchable. Thus, the module is configurable between single and dual channel operation.

3.6.3. Fast Fourier Transform

Many computational problems can be solved in a convenient way in the frequency domain. This, for example, includes pixel binning, deconvolution of the PSF and the Hough Transform (which is presented in section 4.0.2). Therefore, a two dimensional Fast Fourier Transform and its inverse are needed to perform a conversion of the input data into the complex frequency domain and vice versa. The two dimensional transform can simply be achieved by

the execution of a consecutive horizontal and vertical Fast Fourier Transform (FFT). In order to be prepared for the common image sizes which are handled within the *PHI* instrument, a one dimensional FFT/IFFT with configurable transform size between 64 and 2048 points is needed.
As the algorithm and implementation of the FFT/IFFT is well known in literature, *Xilinx* already offers an efficient implementation of the FFT in form of a configurable IP-core which is used to integrate the transform in the architecture of the processing framework.
Basically, the FFT for each point X_k can be calculated as shown in equation 3.10. According to the Cooley-Tukey algorithm [76, 77], this equation can be broken down into a lower part (equation 3.11) as well as an upper part (equation 3.12). The transform with length N can now be expressed recursively by two transforms of length $N/2$. Each of these two terms ca be further divided into an odd and an even part.

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} nk}, \qquad \text{for} \quad k = 0..(N-1) \tag{3.10}$$

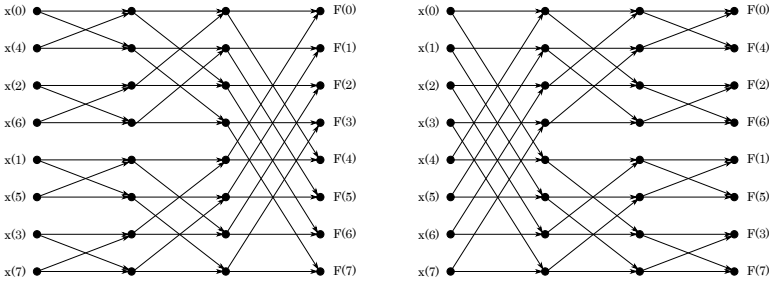
$$\begin{aligned} X_k &= \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N/2} mk} + e^{-\frac{2\pi i}{N} k} \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{1\pi i}{N/2} mk} \\ &= E_k + e^{-\frac{2\pi i}{N} k} O_k \end{aligned} \tag{3.11}$$

$$\begin{aligned} X_{k+\frac{N}{2}} &= \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{2\pi i}{N/2} m(k+\frac{N}{2})} + e^{-\frac{2\pi i}{N}(k+\frac{N}{2})} \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{1\pi i}{N/2} m(k+\frac{N}{2})} \\ &= E_k - e^{-\frac{2\pi i}{N} k} O_k \end{aligned} \tag{3.12}$$

The IP-core provided by *Xilinx* internally uses a fixed-point implementation for complex arithmetic to keep FPGA resource utilization in an acceptable range. Nevertheless, the IP-core offers a single-precision floating-point interface between the stages to avoid overflow issues while achieving similar performance to a full floating-point implementation [78]. Generally, the IP-core offers the following configuration options:

- Burst or fully-pipelined streaming IO architecture
- Variable transform size (run-time configurable), up to 65536
- Forward and backward transform (run-time configurable)
- Radix-2 or radix-4 butterfly
- Input/output format:
  Fixed-point (scaled or unscaled)
  Floating-point (32 bit single-precision, according to *IEEE-754*)
  Block floating-point
- Convergent rounding or truncation after each stage

Due to the high dynamic range, the variant using a floating-point interface was chosen for the *PHI* instrument. Because FPGA resources are still limited, the option for the radix-4 burst IO architecture has been selected. The structure of the IP-core is depicted in figure 3.18. In contrast to the pipelined, streaming FFT which needs $\log_2(N)$ stages, the burst IO implementation only needs one complex radix-4 dragonfly. The transform size has to be configurable between 64 and 2048, which only needs minimum resource overhead.



(a) decimation in time (b) decimation in frequency

Figure 3.17.: FFT structure

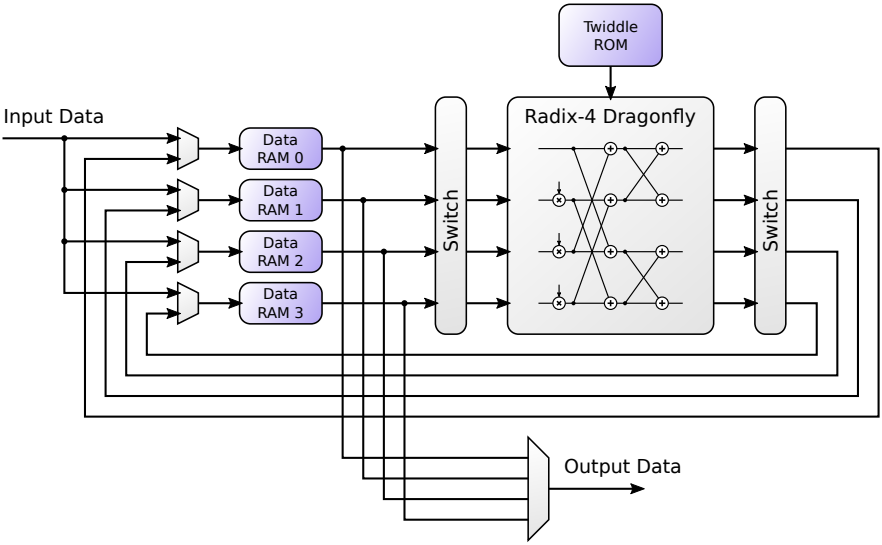
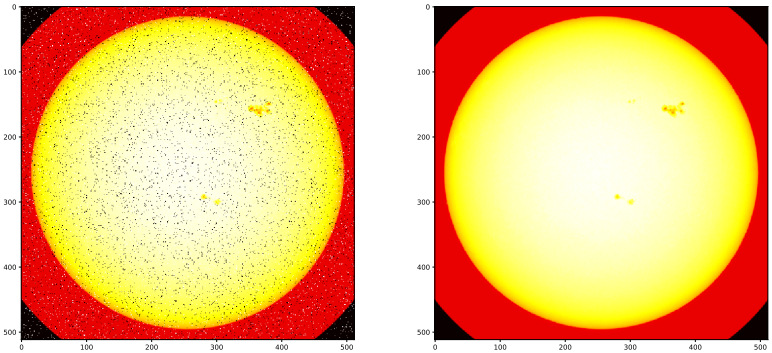


Figure 3.18.: Radix-4 structure of used *Xilinx* FFT IP-Core (according to [78])

3.6.4. Median Filtering

Due to defective pixels on the image sensor, spikes can be observed on acquired input image data. For the effective reduction of this binary noise (also referred as salt & pepper noise), median filtering is a commonly used technique. This method, compared to other filtering methods, is nonlinear and cannot be computed in the frequency domain. Figure 3.19a shows an example image with artificial generated salt & pepper noise. The result of filtering this input with a 3×3 median filter is shown in figure 3.19b.



(a) Image with salt & pepper noise (b) Filtered image

Figure 3.19.: 3×3 Median filtering example

Principle

The median filter takes pixel values from within a filter mask which then are sorted. The median value is selected and the selected pixel is replaced (into a copy of the original image) by this value before shifting the mask to the next image pixel [79]. Figure 3.20 shows the principle in using a 3×3 filter mask. By using this filtering method, edges between plateaus of constant values are being preserved. While for images only containing isolated spikes the median filter with a 3×3 filter mask is sufficient, on images with clusters of faulty pixels a larger filter mask has to be used.

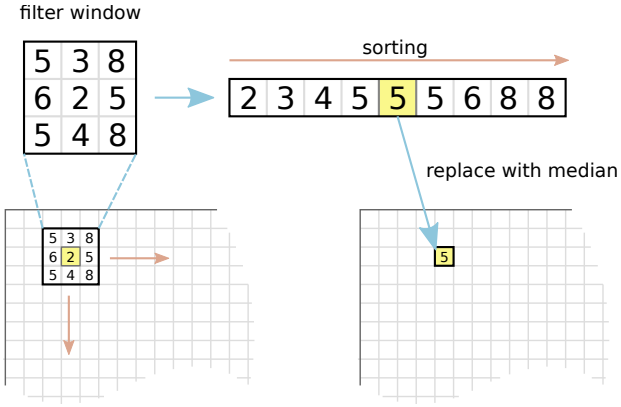


Figure 3.20.: Principle of median filtering

Implementation Considerations

Even though the principle of a median filter is quite simple compared to other filtering methods, some difficulties have to be considered for its efficient implementation in hardware logic. For this reason the hardware implementation of the median filter is depicted in this section.

Because on the example of the *PHI* image processing, acquired image data might contain clusters of defective pixels on its sensor, a median filter supporting both, a 3×3 as well as a 5×5 filter mask has to be integrated. This results in an array of 9, respectively 25 pixel values, each 32 bit wide, which have to be considered for each pixel of an image.

To obtain the median value of this array, the pixel values have to be sorted. A fast and parallel sorting in hardware can be done i.a. using bitonic merging networks [80]. A bitonic merging network is a recursive structure which is able to sort n inputs in parallel. Figure 3.21 shows the recursive structure for sorting $n = 16$ input values (left side). The values are passed through a number of comparator elements, which are represented by the widely used Knuth notation (↕) [81]. Structure `sorter[n]` consists of the structure `merger[n/2]` and two structures of type `sorter[n/2]`, recursively.

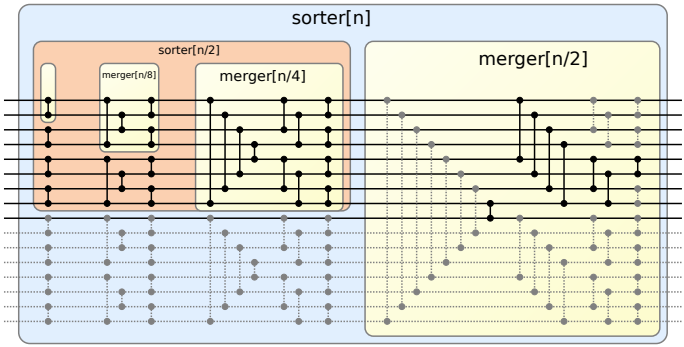


Figure 3.21.: Bitonic sorting network for computing median of 9 input elements

The sorting network can be divided into $n_s = \log_2(n) \cdot (log_2(n) + 1)/2$ stages. Altogether $n_c = n_s \cdot \frac{n}{2}$ comparator elements are needed for fully sorting the input elements. In case only the median is needed, the number of comparator elements can be further reduced (grey in Figure 3.21). This would result in a total count of 32 comparators for a 3×3 median filter and 141 comparators for a 5×5 median filter. While the bitonic merging network has the advantage to be more balanced, the number of comparator elements can be slightly reduced by using an even-odd merge sorting network [82]. Nevertheless, for both methods the asymptotic complexity (C) and depth (S) are

$$C(N) = O(N \log_2(N)) \quad \text{and} \quad S(N) = O(N \log_2(N)) \quad \text{, respectively.}$$

If considering the connection of the median filter to sequential interface of the buffer memory, not all values of the filter window can be read at the same time. Also, the used filter window results in switching between memory pages (which depends on the underlying memory technology) and further reduces the memory data rate. Thus, the above method becomes quite ineffective in this context, as it needs many logic resources.

Based on the given conditions, a more effective method to implement the median filter in hardware logic is to use a structure using adjacent sorting cells [83]. This structure utilizes the fact that only one pixel value has to be sorted into an array which already is presorted. The advantage is that overlapping and previously stored (and already sorted) values are kept in the array when moving the sliding window to the next pixel. The structure, shown in figure 3.22, works similar to a shift register. Each cell includes some control logic and compares new pixel data with its currently stored value and the values of its neighboring cells. If the new pixel value fits in between two stored values, the right-hand registers are shifted (or 'pushed') by one position to the right and the new value is inserted into the free slot.

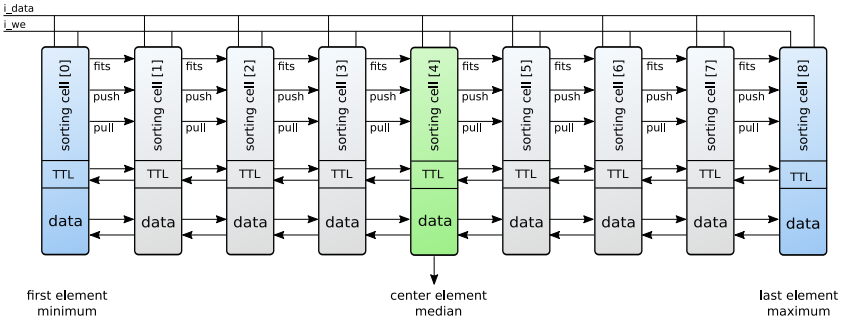


Figure 3.22.: Principle of adjacent sorting cells for median filtering

If once the register chain is initialized with pixel values (in the figure 3.22 on the example of a 3×3 filter window this results in overall 9 cells) the removal of pixel values which are falling out of the sliding window has to be ensured. This is done by counting the time-to-live (TTL) which is initialized for every new pixel value and is decreased with each step. The end-of-life (EOL) of a cell value is reached when this value is zero.
The detailed structure of a single, adjacent sorting cell is depicted in figure 3.23. It mainly consists out of two registers, one for the actual 32 bit pixel value and one to hold the previously defined TTL. Besides these registers there are two multiplexers at the input of each register, some combinational logic

and overall three four comparator elements, whereby three are used for the actual sorting and one is used to check the EOL of the stored value. Therefore, the overall median filter consists of $4 \times n$ comparator elements.
For further timing improvements inside the FPGA, additional register stages can be added in between the cells of figure 3.22. Multiple window sizes can be integrated into one module by generating the structure needed for the largest window size. The median output then has to be taken from a different cell and dedicated cell inputs have to be switched to constant values for small size filter window operation mode. Furthermore, the module needs to generate addresses to read the image pixel values according to the used filter window. Therefore, special considerations about the image borders have to be taken into account.

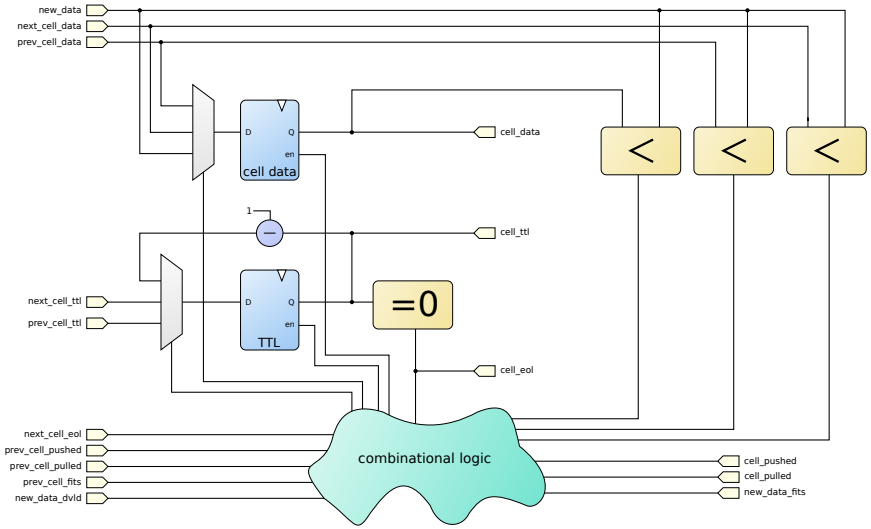


Figure 3.23.: Structure of a single, adjacent sorting cell

3.7. Software Based Self-Test

As mentioned in section 2.4.2 background scrubbing will correct SEUs affecting the configuration layer of the used FPGAs. While this prevents upsets to accumulate over time, it does not prevent actual faults to affect the running user design on the application layer. Faults on this layer have to be mitigated by the user's design. One method for a successful mitigation of these faults is the tool-based application of TMR, which is done for the data acquisition design of the *PHI* instrument. The reason is that data acquisition has to be most reliable because once acquired data is lost, it might not be reproduced. Also, the orbit of *Solar Orbiter* is highly elliptic and data acquisition is most interesting when the spacecraft is close to the sun. Therefore, it is expected that ionizing particle radiation originating from the sun (solar flares) will be higher during acquisition [30]. Another reason why TMR is most suitable for the acquisition is that the design is quite fixed (probably no updates during the mission) and only uses around one third of the *Virtex-4 SX55* resources.
While things for the acquisition design are quite clear, the requirements and constraints for the pre-processing are different in many ways:

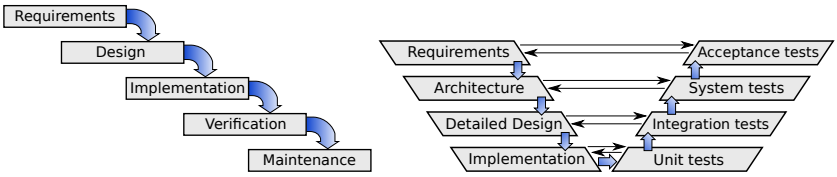
- Processing of safely acquired data can be re-executed any time
- Effects of ionizing particle radiation from the sun to be expected lower during processing phase
- Even though the processing can be split up to multiple FPGA bitstreams, resources are still limited:
  Limited uplink and therefore limited bitsream update capabilities
  FPGA bitstreams will be stored in limited NOR-Flash memory
  (even though storage to the NAND-Flash is possible)

These differences allow, respectively force the pre-processing to use a comparatively lightweight fault mitigation scheme. This scheme basically tries to detect faults during each processing step and when necessary, single processing steps or either more complex processing blocks have to be repeated. This is done by testing each processing module after execution. Therefore, the recently used module is re-executed with a small test vector which is loaded at the end of the buffer memory. Afterward, the checksum of the result will be calculated

in the FPGA to reduce additional data transfer which is finally compared to a pre-calculated golden reference value. In addition, the pre-processing FPGA will receive an asynchronous reset before the execution of each processing operation.

3.8. System Verification

With the growing complexity of software and also hardware systems, testing is a very important key factor in the development of electronic systems, especially in the field of the aerospace industry. Many examples in the past have shown that little failures in hardware or software can lead to serious complications or even a complete loss of a mission. During the first launch of the European *Ariane 5* launcher carrying two satellites of the *Cluster* mission, a software failure in the Inertial Navigation System (INS) lead to a fatal course deviation of the spacecraft and therefore the total loss of its payload and the mission. Throughout a later investigation of the incident it was found that the reason was an arithmetic overflow on a conversion from a 64-bit floating-point value into a 16-bit signed integer variable inside the software of the INS. Finally, the mistake could have been detected by more extensive testing of the system.



(a) Waterfall Model (b) V-Model

Figure 3.24.: Sequential models for verification

During software development, for example, usually the effort needed for intensive testing at least is around 25%. For development of electrical systems (including hard- and software components) different methods for assurance of the quality are typically used. Two of these models, the Waterfall Model and the V-Model, are depicted in figure 3.24 [84].

In the simplest case, the sequential model is represented by the *Waterfall Model* shown in figure 3.24a. After the refinement from the requirement specification down to the implementation, the design is tested against the requirements. As a drawback of this model is that it is not intended to redo a certain step a second time, even if tests are added afterwards. Also maintenance after delivery is only envisaged in a rudimentary way. In contrast, a more fine grain integration is provided by the *V-Model* depicted in figure 3.24b. This model is commonly used and links each design level to an appropriate test level.
The *V-Model* is also used during the development of *PHI*. For verification, the presented processing framework has to be tested against the requirements on multiple layers. This includes test-benches for unit tests in simulation on Register Transfer Logic (RTL) level, automated testing of all implemented processing functions within the DPM and verification and evaluation of the complete processing chain and its final products. A design-flow using hardware and software co-simulation in combination with a *LEON* processor, as presented in [85], would also be possible in principle but was not applied to the *PHI* DPM. A general overview of testing the processing framework for *PHI* will be given in the next subsection. Also, a dedicated test framework for the verification on the various levels will be presented.

3.8.1. Testing of the PHI Processing Framework

Although, neither the *PHI* processing pipeline nor the *PHI* DPU itself could lead to injury of the spacecraft and a total loss of the *Solar Orbiter* mission, a damage of the *PHI* instrument or a degradation in terms of availability is undesired due to the high costs of the scientific payload and the many person-years involved in this sophisticated project. Because upload data rate is very limited and updates are only possible at certain times during the mission, the instrument's software and specially the FPGA configurations have to be tested carefully. These FPGA bitstreams can not be patched like the software because small changes in the design will lead to significantly different results during place and route of the FPGA design. Even when compressed, a bitfile still has the size of around 800 KiB.

The processing architecture offers the ability to rearrange configurations with the needed operation modules. Even new modules can be added and the resulting design can be uploaded to the spacecraft during the mission. These possibilities allow to react to changing mission objectives with flexibility but also make it essential to test the generated set of bitfiles in an automated way.
For autonomous testing of the developed processing architecture a framework is needed which allows to effectively supply the DPU with suitable test vectors and allows a calculation of reference functions without a high programming effort. Especially for the implementation of reference models for the image processing pipeline a toolkit such as the widely used *MATLAB/Simulink* or the open source alternative *Octave* would offer easy to use support for large matrices and data visualization. In the scientific space sector (e.g. in astrophysics), the *Interactive Data Language* (*IDL*) is widely common. For example, *IDL* is being used for evaluation of data collected during the *STEREO* mission.
Meanwhile, specially in the area of scientific computing, the *Python* programming language gets more and more popular. It is a dynamic type language supporting the object-oriented programming paradigm. Besides a large and comprehensive standard library, the official repository for third-party *Python* extensions (the *Python Package Index*) contains a large number of packages with a wide range of functionality. This includes e.g. numerical mathematics, graphical user interfaces, visualization of data and import/export functionality for different file formats [86].
Because of its high productivity, the good establishment and the wide range of existing packages it was decided to use *Python* to implement an extensive framework to test and evaluate the processing on the *PHI* DPU.

The principle test setup is depicted in figure 3.25. The Electrical Ground-Support Equipment (EGSE) mainly consists of a host PC running a *Linux* or *Windows* operating system with an installation of *Python3* including the EGSE test framework software and the needed dependencies. The Design Under Test (DUT), in this case the *PHI* DPM, is connected via an external Ethernet to *SpaceWire* bridge using the TCP/IP protocol. The EGSE is also connected to an external power supply (either by USB/serial interface or Ethernet) to safely power the DUT and monitor the power consumption. Furthermore, the host PC of the EGSE may contain dedicated hardware (e.g. a PCI card with

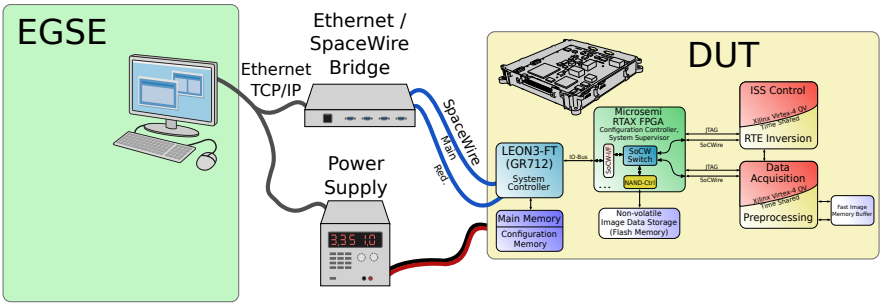


Figure 3.25.: Test setup using Ethernet to *SpaceWire* bridge

FPGA devices) for testing of proprietary interfaces. This, for example, can be dedicated serial communication interfaces (derivated from SPI) or interfaces for transfer of image data between the sensor and the DPM (e.g. *ChannelLink* interface).
With the exception of the use of the integrated PCI hardware, the host PC not necessarily needs to be part of the EGSE. Moreover, the test software can be run from any other computer (also via the Internet) which gives more flexibility in terms of testing the already delivered DUT remotely.
After powering the DUT, the software framework usually will supply the latest needed system test software to the DPM's bootloader which is started subsequently. Because the bootloader is a critical part of the system software and might already be programmed into a fixed boot PROM, the communication during the boot process is using CCSDS packet format. In contrast, communication and commanding of the system test software via *SpaceWire* is done using a proprietary packet format. This has the advantage of minimizing the communication overhead during test. For the purpose of background debugging, even when the system software is not responding, the memory (and also the IO and configuration area) can be read and written through the RMAP protocol.

An overview of the communication between the test software and the DUT is given in figure 3.26. Besides the communication during the boot process and the memory access via the RMAP protocol, the test framework

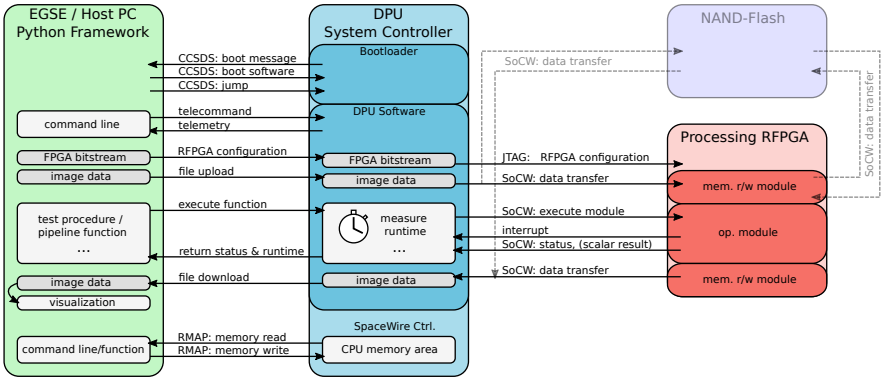
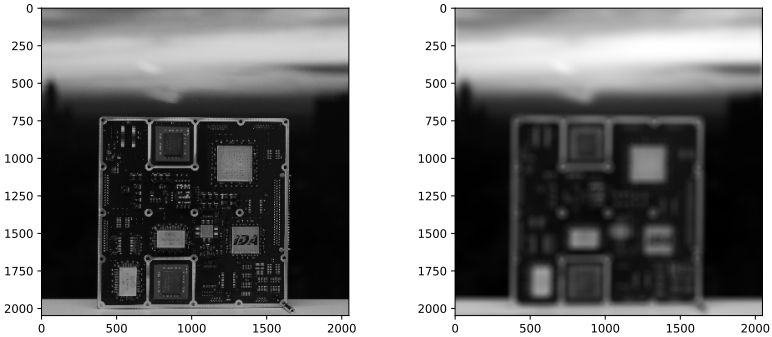


Figure 3.26.: Data flow through the test setup

is able to initiate data transfer between the host PC, the System Controller, the NAND-Flash memory and the FPGAs. It can also configure both of the *Virtex-4* FPGAs and start the execution of functions of the data processing framework. All the functions from the on-board software part of the processing framework are mapped to similar function calls in the *Python* test software. These function calls are blocking until the associated hardware module of the processing framework has finished its execution. This is used to invoke several processing steps sequentially, forming more complex processing procedures. Furthermore, the time difference between the start of a processing function and its return is determined in the software running on the DPM and passed to the test framework on the host PC. The resulting run-time not only includes the pure execution time of the hardware module. It also contains the time needed to set up module parameters and read out interrupt and module status information via the SoCWire interface. All these run-time statistics are finally collected and logged by the EGSE software in the background. Therefore it enables a fine-grain, easy-to-use run-time evaluation of more complex processing routines.



(a) Original image (b) Filtered image

Figure 3.27.: Result of FFT low-pass filtering example

Besides the direct setting of parameters and status readout via the command line, the main task of the *Python* framework is to run dedicated functions from within a test suite or to execute and evaluate more complex processing pipelines. While the test functions will compare the results of the processing functions either against a *Python*-based (bit accurate) reference implementation or pre-calculated values, the pipeline functions help to implement and evaluate more complex processing procedures before implementation into the flight software. These procedures are formed by the sub sequential execution of different processing functions, RFPGA reconfiguration and transfer of image data. Two examples of such pipeline procedures have been presented in sections 4. A break-down analysis of module/function run-time, which is done with the help of the test software, will be presented in section 5.1.
For illustration how a sequence of processing functions is mapped into the *Python* framework, a simple example is depicted in listing 3.1. The calculation of a low-pass filtering of an image in the frequency domain on the *PHI* DPM is carried out and plotted within only a few lines of code inside a *Python* functions. The plotted input and output images are shown in figure 3.27.

```
def fft_filt(self):
    p = self.preproc
    image = Image()
    image.load('image.png', imsize=[2048, 2048])

    image_mask = Image()
    image_mask.load('filter_mask.png', imsize=[2048, 2048])

    # Upload and store image
    p.store_direct(image, 0x00000)

    # Upload and store mask
    p.store_direct(image_mask, 0x200000)

    p.real2fpcplx(0x00000, 0x0400000, 0x400000)

    p.fp_fft(0x0400000, 0x1400000, 2048, 2048)
    p.fp_fft(0x1400000, 0x1800000, 2048, 2048, vertical=True)

    p.fp_cmult(0x1800000, 0x1000000, 0x1c00000, 0x400000)

    p.fp_ifft(0x1c00000, 0x2000000, 2048, 2048, vertical=True)
    p.fp_ifft(0x2000000, 0x2400000, 2048, 2048)

    p.fpcplx2real(0x2400000, 0x2800000, 0x400000)

    # Load FFT result...
    data_fft = p.load_direct(0x2800000, 0x200000)

    image_fft = Image()
    image_fft.set_image(data_fft, [2048, 2048])

    image_fft.show_re()
```

Listing 3.1.: *Python* example code invoking the FFT for image filtering

4. Proof of Concept

To achieve a fast realization of regular scientific processing of the acquired image data, a defined set of basic and specialized processing has been implemented. Especially the further specialized functions needed for on-board and in-flight instrument calibration seem to be quite complex. At first glance, there seems to be no way around implementing these functions as dedicated hardware modules, each. When taking a closer look, most functions can be broken down to already existing modules.
As a proof of concept, this section will depict the regular pre-processing as well as two functions which are needed for in-flight flat field calibration of the Full-Disc Telescope: the circular Hough Transform and the Kuhn-Lin-Loranz algorithm. While the Kuhn-Lin-Loranz algorithm [70] is an iterative approach of calculating the gain parameter for each individual pixel of the sensor out of a set of displaced images (in this case overall nine images), the Hough Transform is needed to determine the exact center position of the solar disk in each of these images. This information is needed by the Kuhn-Lin-Loranz algorithm.
To use the ability of FPGA reconfiguration as best as possible, the processing modules have to be arranged into several sets in the form of configuration bitstreams. This arrangement might be different for other projects and might also change for the final implementation of the *PHI* DPU. For the work presented in the following sections, a constellation of overall three FPGA configurations shall be assumed. The contents of these sets are depicted in table 4.1.

Table 4.1.: Arrangement of configuration bitstreams

Configuration 1	Configuration 2	Configuration 3
op_addsub	op_fp_cext	op_float2fix
op_mult	op_fp_fft	op_fix2float
op_div	op_fp_mult	op_log
op_thold	op_crop	op_sqrt
op_crop	op_median	op_hist
op_shift		op_logic
op_multisum		op_binning
op_logic		op_tmr

4.0.1. Regular Pre-Processing Pipeline

The principle pre-processing of the acquired image data has already been briefly presented in section 3.2.2. As the instruments performance strongly depends on the environment, the actual configuration of processing steps will be determined during in-flight commissioning and calibration of the instrument. These several steps are already pre-defined in form of on-board software procedures, involving the necessary processing modules [87]. As a proof-of-concept, the pre-processing pipeline from section 3.2.2 will be analyzed including all important parts of the processing, even they might not be used within the regular mission. For this regular pipeline, the individual steps can be further broken down into the execution of processing modules as shown in figure 4.1:

① **Preparation of input images** This step a set of 24 images from the NAND-Flash to the SDRAM buffer memory. For the detection of corrupted pixels, the absolute value of the derivative is calculate for horizontal and vertical direction of each image. A binary mask is generated in dependency of a parametrizable threshold. Affected pixel values are replaced by the NaN value.

② **Dark and flat field correction** Dark and flat field images are loaded from NAND-Flash and stored into the buffer memory. After the dark field is subtracted from the input data set, the 24 input images are each divided by the corresponding flat field image. The calculation is according to (3.2).

③ **Local median filtering** Corrected images are filtered by the 5×5 median filter. Subsequently, the corrupted pixels marked as NaN in the original images during ① are each replaced by its filtered filtered pendant. This step requires a few reconfigurations of the FPGA.

④ **PSF deconvolution** This step deconvolutes the PSF of each image by multiplication with the Wiener filter in the frequency domain.

⑤ **Polarization demodulation** The input data is converted into the four Stokes parameters according to (3.6). The matrix multiplication is broken down into four subsequent multiplications and three additions.

⑥ **Preparation for RTE** Before data is finally transferred to the inversion of the RTE which is carried out on the first of the two RFPGAs, the demodulated data is converted into floating-point format and reordered.

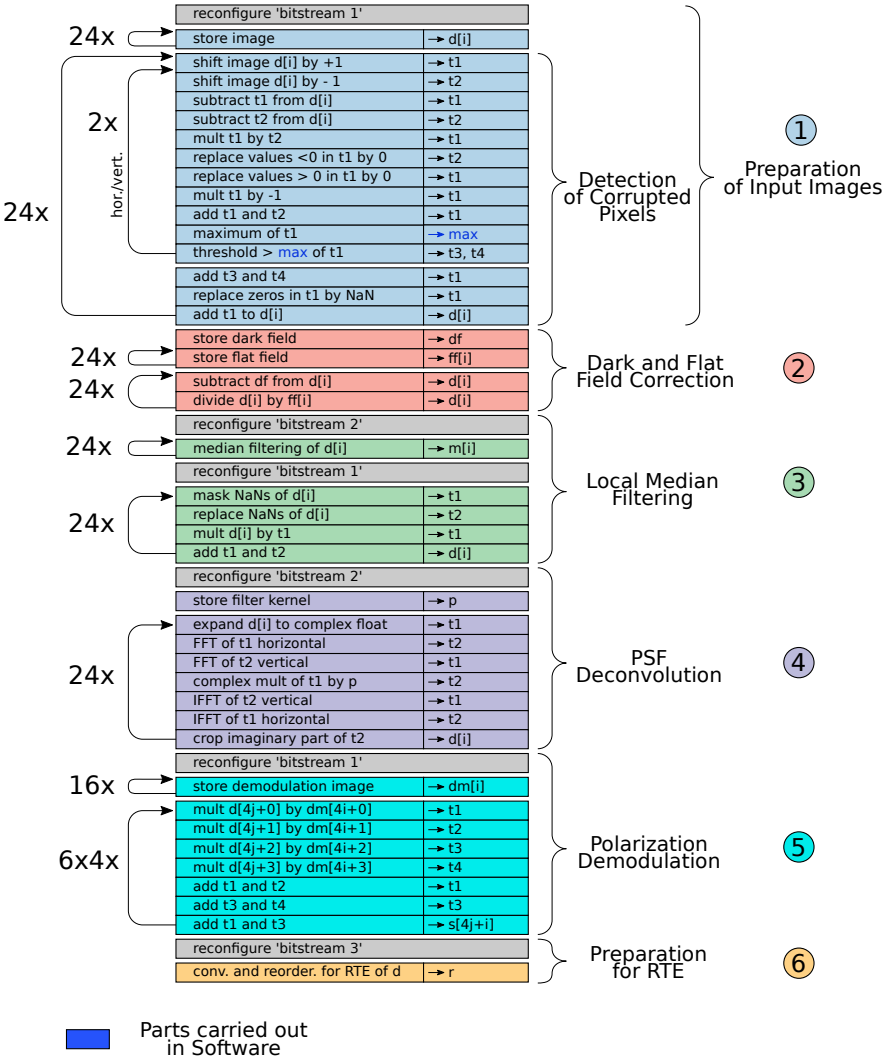


Figure 4.1.: Example regular pre-processing within the framework

4.0.2. Hough Transform

The Hough Transform [88–90] is a well known technique in computer vision and image processing for the feature detection of parameterizable geometric objects. In case of the flat field calibration of the *PHI* instrument it is used to detect the center position of the solar disc.
Generally, the Hough Transform is applied to binary images. Therefore, gray-scale input images have to be pre-processed before the actual to detect the edges of the solar disk. This is done by calculating the gradient of the image followed by a thresholding. These steps can be further divided and mapped to the set of processing modules/functions (see table 3.3):

- Calculate gradient:
  - ① 2× shift image by one pixel in hor./vert. direction
  - ② 2× subtraction of shifted image from original
  - ③ 2× calculate square of hor./vert. gradient
  - ④ Add these two results to final gradient
- ⑤ Calculate threshold to get binary image

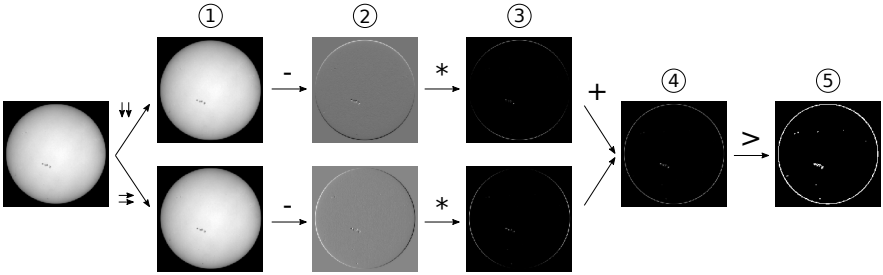


Figure 4.2.: Binarization of input images

The corresponding processing steps for the binarization of the input image are depicted in figure 4.2. After this pre-processing is done, the actual Hough Transform is carried out on the binary image.

The conventional circular Hough Transform uses an accumulator array to find the center of a circle with a given radius. Each point (x, y) in the original image is used to define a circle with the center (x, y) and the given radius r in the Hough parameter space. In the accumulator array, each cell through which this circle is passing is then incremented by one. The circular center can finally determined by finding the local maximum point in the accumulator array. For the case the radius is not exactly known, the transform has to be iterated over possible radii, leading to a three dimensional parameter space and accumulator array.

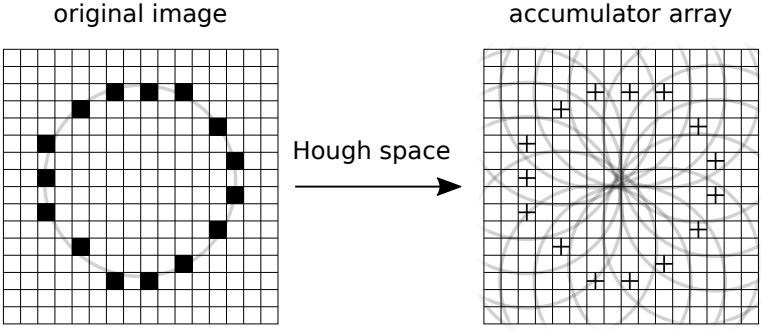


Figure 4.3.: Principle of conventional Hough Transform

Implementing the algorithm of the circular Hough Transform as dedicated hardware module would be possible but also very specific and complex. Since this procedure is very similar to a convolution with a circular filter kernel, it has been shown that the circular Hough Transform may also be performed by a complex multiplication in the frequency domain (see [91, 92]). As in the case of the *PHI* instrument the solar disc center is always assumed to be within the image, no further padding is required. Therefore, the already existing modules of the Fast Fourier Transform (and inverse) and complex multiplication can be used instead.

For finding the centers of the overall nine images, all these processing steps have to be combined into one sequence. A sequence which uses reconfigurations of the FPGA which also copes with the limited buffer memory of one GiB is shown in figure 4.4.

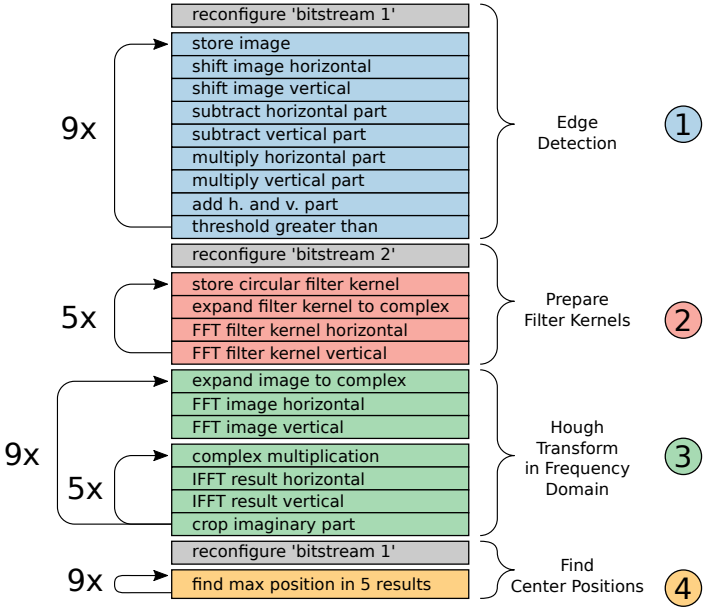
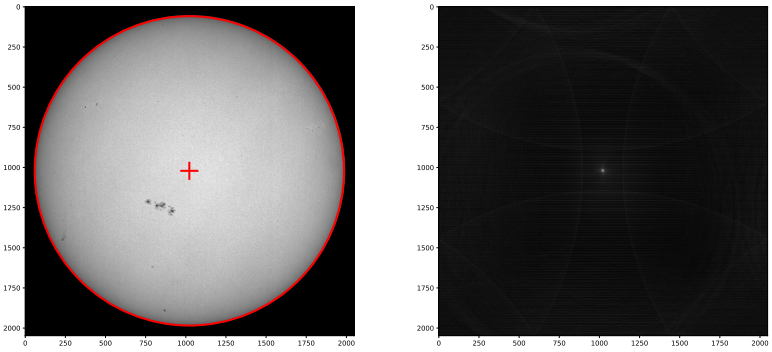


Figure 4.4.: Hough Transform for 9 images using reconfiguration

The implementation makes use of FPGA reconfiguration three times during the processing of all the nine input images. This is based on the previously listed arrangement of processing modules in table 4.1. Therefore, the edge detection is split into two parts: one including the horizontal/vertical shifting of the input images and one including the arithmetical operations. Before the actual Hough Transform is carried out in the frequency domain, several circular filter kernels for different assumed radii have to be prepared. These images can either be calculated on demand on the CPU and transformed into the frequency domain inside the processing FPGA or calculated in advance and stored onto

the NAND-Flash memory board (either spatial or frequency domain). In case of the *PHI* instrument, filters are constricted to a range of five different radii. The Hough Transform is computed for all nine images by a multiplication of each image with all five filter images in the frequency domain followed by an inverse Fourier-Transform of each result. This leads to overall $9 \times 5 = 45$ images demanding overall $720\,\mathrm{MiB}$ of buffer memory, while the input images ($9 \times 16\,\mathrm{MiB}$) and the complex filter images ($5 \times 32\,\mathrm{MiB}$) already require $304\,\mathrm{MiB}$. Therefore, the computation has to be optimized to overwrite obsolete input data to avoid additional reconfiguration of the FPGA. Finally, each of the nine sets of accumulator images is scanned for its maximum position which is used to calculate the circular center of each image.

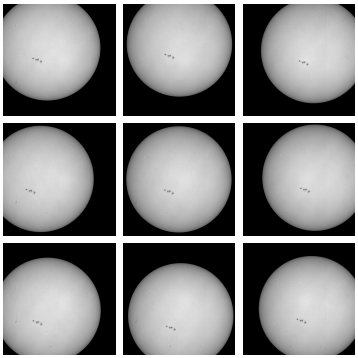


(a) detected center of the solar disk (b) maximum peak in hough space

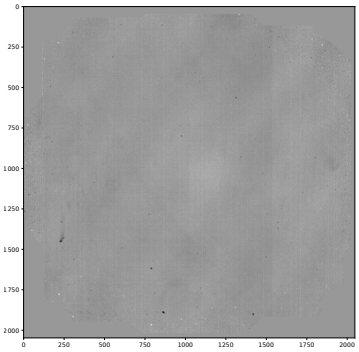
Figure 4.5.: Detecting the center of the solar disk through Hough Transform

4.0.3. Kuhn-Lin-Loranz Algorithm

For in-flight acquisition of flat field calibration data for the *PHI* instrument, an algorithm according to Kuhn, Lin and Loranz is used [70]. The method is used to overcome the need of a spatially uniform light flux in front of the detector. Therefore, the algorithm calculates the individual gain for each detector pixel using a set of displaced input images. On the example of the *PHI* instrument, overall nine different input images of the solar disc will be used for calibration (figure 4.6a).
Generally, the algorithm works on the logarithm representation D_i of each corresponding input image d_i. It consists of an iterative part as shown in equation 4.1 which calculates the individual pixel gain G. The equation includes a constant K which is calculated according to the term in 4.2. The displacement of the input images, which has been determined in advance using the previously mentioned Hough Transform, is represented in these terms as a_i for each image i. The algorithm will sum up over all possible combinations of the input images ($i < j$). In case of the nine input images used for *PHI*, the number of valid combinations is 36. Pixels with a signal strength which is under a certain threshold will not be included in the calculation. Therefore, the result has to be scaled by $\frac{1}{n(x)}$, which accumulates the valid pixels.



(a) displaced input images



(b) calculated flat field

Figure 4.6.: Input and result of the KLL algorithm

$$G^{r+1}(x) = K(x) + \frac{1}{n(x)} \sum_{i<j} [G^r(x - a_i + a_j) + G^r(x - a_j + a_i)] \quad (4.1)$$

with

$$K(x) = \frac{1}{n(x)} \sum_{i<j} [[D_i(x) - D_j(x - a_i + a_j)] + [D_j(x) - D_i(x - a_j + a_i)]] \quad (4.2)$$

The presented algorithm is quite computation intensive. As it take multiple images and offsets into account, a dedicated *VHDL* design would include complex address generation and would be quite static related to possible future changes. Instead of a dedicated processing module the algorithm can be broken down and mapped to the available, basic processing operations. This would preserve flexibility and keep the implementation of this algorithm simple while achieving reasonable runtime.

The resulting flow of executed processing modules and functions as well as FPGA reconfigurations (for arrangement of modules as assumed in table 4.1) is shown in figure 4.7. Generally, the algorithm can be divided into five steps:

① **Preparation of input images** The overall nine input images are loaded into the buffer memory. Corresponding to the input, nine masks are calculated to indicate which pixels are over a defined threshold. Also, the input images are transferred to a logarithmic scale by calculation of $\log_2$ and removal of invalid (NaN) results.

② **Calculation of initial K** During this step the initial K is calculated according to (4.2). Therefore, the scaling factor $\frac{1}{n(x)}$ is calculated first. It includes the overall number of combinations which have an influence on a specific pixel-position of the result. As the calculated K can be considered as the initial value G_1 of the following iteration, it is copied to a separate memory position.

Iteration of G This step is the actual iteration of the pixel-gain G.

③ **Calculation of G** This step includes similar shift and mask operations as the calculation of K. In contrast, it does not include the subtraction.

④ **Correction of G** The correction of G also includes the division by $n(x)$, which already has been summed-up during the calculation of K. In this example, it is assumed that the algorithm converges after five successive iterations.

⑤ **Exponentiation** The transfer, back from the logarithmic scale, only needs to be calculated for the single resulting flat field instead of the nine shifted input images. Therefore, the calculation of $g(x) = e^{G(x)}$ is carried out in the on-board software. This calculation is assumed to take approximately 40 s on the *GR712* plus additional time for transfer of the image.

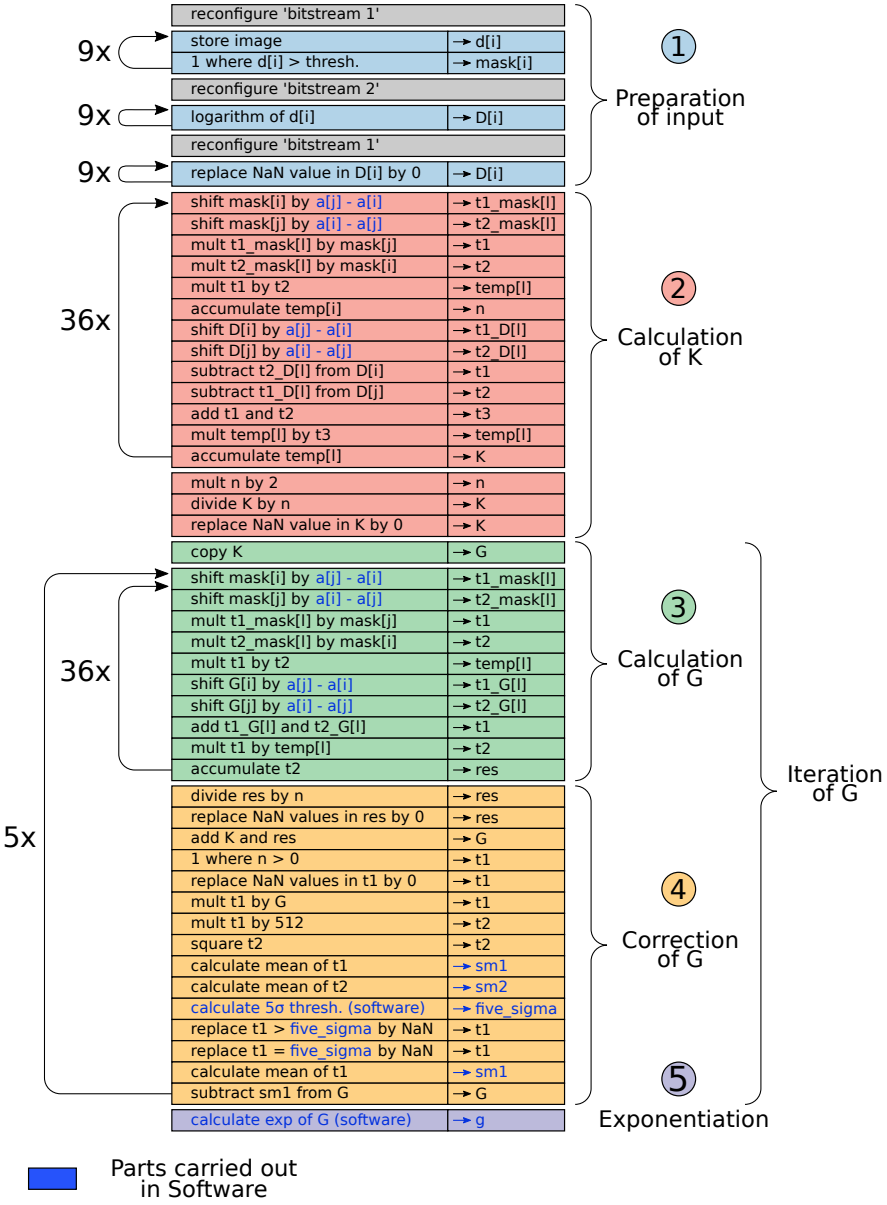


Figure 4.7.: KLL flat field calculation within the framework

5. Evaluation and Comparison

In this chapter, the introduced image processing framework shall be evaluated in terms of processing speed, power consumption and FPGA resource utilization. The processing performance will be interpreted in a general per-module basis in section 5.1.1. The measured results will be compared against the predicted performance. Also, the run-time will be compared to the available components of a pure software implementation for the *GR712*.
As for the per-module analysis only considers the pure run-time of the hardware implementation, section 5.1.2 additionally analyzes the overhead of the software based self-test introduced in section 3.7. This overhead is then taken into account for the examination of the run-time in the context of the three use-cases which have been presented in chapter 4 [93].
Finally, section 5.2 will present measured power consumption of the processing framework on a per-module basis. Section 5.3 will shortly summarize the resource usage of the implemented modules in the context of the used *Virtex-4* FPGA.

5.1. Processing Performance

The following subsections will present the measured run-time of selected functions as well as the overall processing time required by the processing examples presented in section 4. The measurements have been done using the *Python* software test framework which has been presented in section 3.8.1. Besides these practical measurements, section 5.1.1 also covers some theoretical aspects to explain, validate and classify the measured quantities.
The following sections also compare the performance of the presented processing framework against a software based reference implementation. This reference implementation has been implemented by *MPS* as a backup solution.

To achieve equal conditions as best as possible, the reference implementation is executed on the *GR712* CPU, which is also part of the *PHI* DPM. Both, CPU and FPGA implementation, are running at the same clock speed of 50 MHz. The *GR712* CPU also uses the same type of SDRAM memory which is attached to the second RFPGA, albeit only one instead of four memory stacks can be connected to the CPU. The flexibility of the FPGA also allows to use a bus width of 64 bit compared to the 32 bit bus width of the *GR712*.

5.1.1. General Processing Performance

In order to analyze the behavior of the implemented modules/functions of the processing framework in dependance of the involved image size, the run-time of these functions has been measured with the help of the *Python* test framework as described in section 3.8.1. The result of these automated measurements are plotted in figure 5.1. The horizontal axis of the plot covers the range of the involved image size up to $134\,\mathrm{Mibit}$ for an image of the maximum size of 2048×2048 pixels. The vertical axis represents the run-time in seconds.
Generally, a perfectly linear dependency between involved image size and run-time can be concluded. However, while the run-time of some functions seem to be in a similar range, a few functions show significant differences. These differences can be explained by having an insight into the technical details of the implementation of the processing modules and the memory interface. Generally, the functions can be classified into the following categories:

Memory interface limited In this case the throughput of the module is limited by the external memory interface. This category can further be subdivided into **triple-port**, **dual-port**, **single-port** and **window-operator** access types. Modules of this class would benefit from speeding-up the memory data rate. Examples are addition/subtraction as well as the median filtering.

Processing limited Modules of this category are limited by its own processing performance. A good example is the implemented $\log$ module, which is internally implemented using a CORDIC IP-core and needs several clock cycles for calculation of the result.

Combination In some cases the behavior of the module strongly depends on its parametrization and can either be limited by its processing performance or the external memory interface. Example of this category is the used implementation of the FFT.

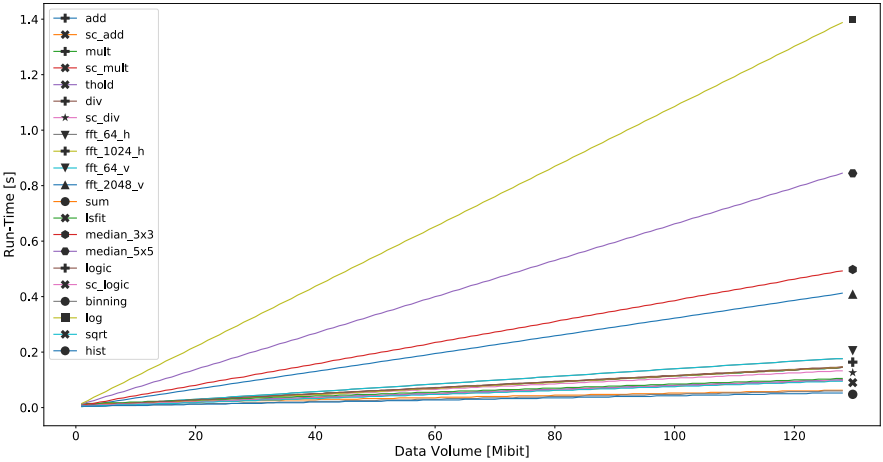


Figure 5.1.: Run-time of functions implemented in hardware

A selection of processing functions is depicted in table 5.1. It shows the measured run-time, the resulting throughput, as well as the predicted throughput. The run-time is related to the size of an involved image with the size of 2048×2048 pixels. This reflects the typical use-case throughout the processing pipeline. It should be noted that this involved image size is not necessarily the amount of input or output data to the function. It also has to be considered that functions intended for complex pixel format have to process twice the amount of data for the same involved image size.
The metric for the throughput, given in $\mathrm{Mibit/s}$, is also related to involved image size. In contrast to measured run-time, it considers the pixel format ($32\mathrm{bit}$ real-only or $64\mathrm{bit}$ complex) of each function. Table 5.1 shows the measured throughput as well as the predicted throughput for each listed function. The estimation of the function run-time and throughput based on the implementation

of the functions and the memory interface is detailed in appendix A. As can be observed, the difference of the measured values to the predicted values is less than 10 % (with the exception of the sum) and therefore, the measurements can be considered to be plausible.
In comparison to the results measured and predicted for the hardware-based framework, the measured results for a software reference implementation of the functions are shown in figure 5.2. As mentioned before, this software implementation is intended as a backup solution running on the *GR712* processor of the *PHI* DPU and was implemented by *MPS*. The *GR712* is running at the same clock speed as the RFPGA and also uses the same type of memory, albeit with only half of the bus-width.
Even though the software implementation might have space for optimization and the results should be regarded with care, the speed-up of the functions lies within multiple orders of magnitude.

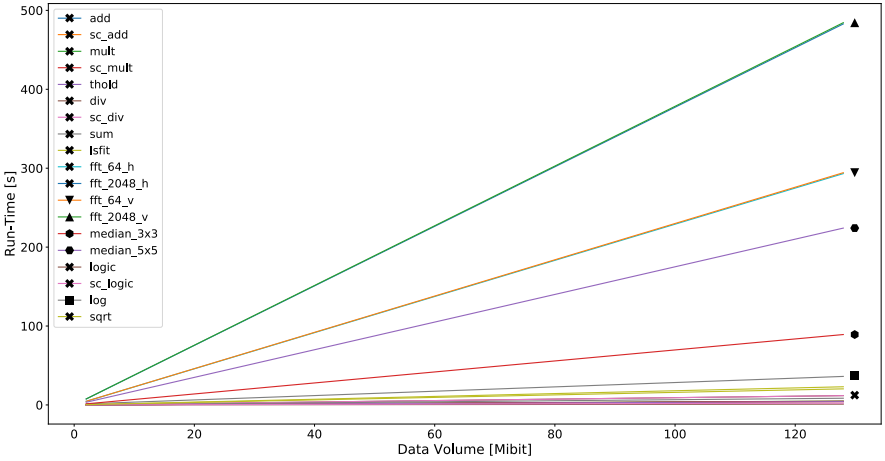


Figure 5.2.: Software Implementation

Table 5.1.: Throughput of typical processing functions

Function	**Run-Time** 2048×2048 [s]		**Throughput** [Mibit/s]		
	Framework	Software	Measured	Predicted	Software
Memory interface limited, single-port					
`sum`	0.063	8.5	2031.7	2465.1	15.1
`hist`	0.052	102.2	2461.5	2465.1	1.3
Memory interface limited, dual-port					
`sc_add`	0.099	4.1	1299.5	1311.9	31.0
`sc_mult`	0.100	4.7	1286.4	1311.8	27.1
`sc_logic`	0.099	2.3	1292.9	1306.9	56.3
`sqrt`	0.097	23.0	1319.6	1306.9	5.6
`thold`	0.099	4.2	1292.9	1311.8	30.4
`lsfit`	0.105	20.4	1219.0	1316.8	6.3
`crop`	0.190	0.7	673.7	658.4	174.1
`binning`	0.061	18.6	2081.3	1992.0	8.9
`shift`	0.193	3.9	663.2	658.4	32.8
Memory interface limited, triple-port					
`add`	0.145	4.4	879.7	896.0	29.2
`mult`	0.146	4.9	873.7	896.0	26.2
`div`	0.147	11.8	867.8	896.0	10.8
`logic`	0.145	2.8	879.7	893.6	45.3
Memory interface limited, vertical					
`fft_2048_v`[1]	0.821	482.8	311.8	419.7	0.5
Processing limited					
`log`	1.389	39.7	92.1	101.7	3.2
`sc_div`	0.134	11.8	955.2	1017.3	10.8
`median_3x3`	0.493	89.2	259.9	254.0	1.4
`median_5x5`	0.847	224.1	151.2	152.0	0.6
`fft_64_h`[1]	0.356	293.1	716.2	728.8	0.9
`fft_1024_h`[1]	0.291	445.4	879.8	900.1	0.6
`fft_64_v`[1]	0.356	293.1	716.2	728.8	0.9

[1]The FFT-sizes have been selected according to the min. and max. run-time, each for horizontal and vertical orientation.

5.1.2. Self-Test Overhead

The framework includes the ability to automatically execute a self-test based on test-vectors after the execution of each function. For sufficient test-coverage (e.g. finite state machines, address counters and reset logic, FIFOs, etc.), only small test vectors are needed. This also applies to the used architecture of the FFT, as the control flow is independent of the used transform size or configured direction. Since the same length of these test-vectors is used for all implemented functions, the induced run-time can be considered as constant. Table 5.2 lists the run-time for a typical 2048×2048 pixel image, each for enabled and disabled self-test. The resulting overhead, given in percent, depends on the image size and is negligible for large amounts of data to be processed. Especially for fast functions the overhead can be around 35%.

Table 5.2.: Run-time overhead with enabled self-test

Function	**Run-Time** 2048×2048 [s]		**Overhead** [%]
	Self-Test enabled	Self-Test disabled	
`add`	0.168	0.145	15.9
`add_sc`	0.120	0.099	21.2
`sub`	0.168	0.145	15.9
`sub_sc`	0.120	0.099	21.2
`mult`	0.168	0.146	15.1
`mult_sc`	0.122	0.100	22.0
`thold`	0.122	0.099	23.2
`div`	0.170	0.147	15.6
`div_sc`	0.155	0.134	15.7
`real2fpcplx`	0.205	0.183	12.0
`fpcplx2real`	0.195	0.173	12.7
`fft_64_h`	0.189	0.178	6.2
`fft_1024_h`	0.156	0.145	7.6
`fft_64_v`	0.189	0.178	6.2
`fft_2048_v`	0.421	0.410	2.7
`fp_cmult`	0.310	0.288	7.6
`fp_cmult_sc`	0.213	0.190	14.7
`sum`	0.080	0.063	27.0
`lsfit`	0.122	0.105	16.2
`median_3x3`	0.515	0.493	4.5
`median_5x5`	0.868	0.847	2.5

Table 5.2 – continued

logic	0.168	0.145	15.9
logic_sc	0.120	0.099	21.2
shift	0.218	0.193	13.0
binning	0.083	0.061	36.1
log	1.410	1.389	1.5
sqrt	0.117	0.097	20.6
histogram	0.074	0.053	42.3

5.1.3. Regular Preprocessing

The particular steps of the regular processing pipeline have been presented in section 3.2.2 and further detailed in section 4.0.1. As the run-time for each function can be obtained from table 5.2 or A.3, the overall time needed to run the complete processing can be calculated. Furthermore, the run-time can be broken-down in the particular steps listed in figure 4.1. The result of this analysis is shown in figure 5.3.

The total run-time of the regular processing is expected to be 370 s, as shown in the center of the diagram. The center ring represents the coarse-grain categories from section 4.0.1 (with respective numbers), whereas the outer ring includes the fine-grain functions which are executed.

As is apparent in the outer ring of the diagram, especially the transfer of the input data-set and the calibration data from the NAND-Flash memory already takes a significant amount of time compared to other functions. The reason for the limited data-rate during the transfer is not the presented processing framework. It is rather limited by the the NAND-Flash module and its controller which has to deal with restricted resources on the System-Supervisor FPGA. In contrast, the overall amount of time needed for the necessary reconfiguration of the FPGA is relatively short and therefore negligible.

Considering only the available software equivalents from table 5.1, the calculation for one single data-set would take about 15 hours, while most of the time is spent during the Fast Fourier Transform of the images.

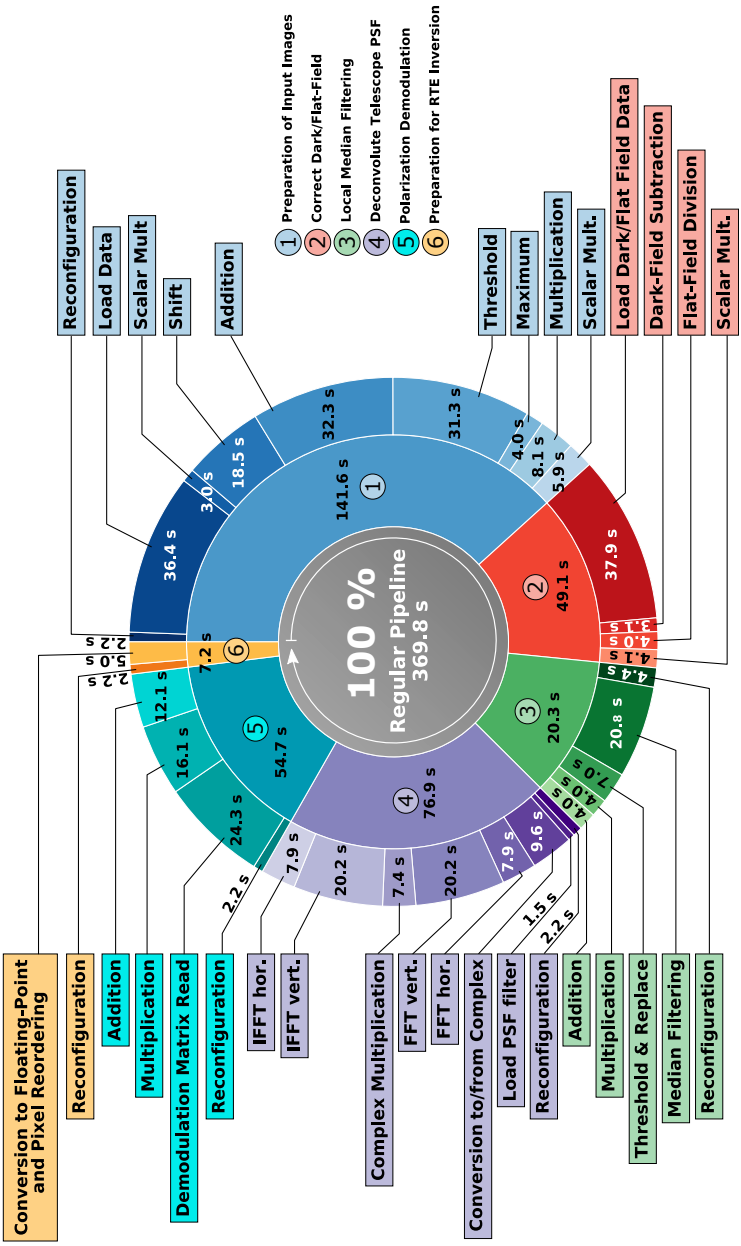


Figure 5.3.: Breakdown analysis of regular pre-processing

5.1.4. Hough Transform

As done for the regular pre-processing, also the overall run-time of the implemented Hough Transform for nine images (see section 4.0.2) can be dissected into its basic elements. This analysis is illustrated in figure 5.4. The several steps are related to figure 4.4. As the result only consists of the maximum positions of each result, no image data has to be transferred back neither to the NAND-Flash memory nor to the processor.

As shown in the figure, the Hough Transform is expected to take around $130\,\mathrm{s}$ for the complete set. The implementation spends most of its time during execution of the FFT or its inverse. As the buffer memory can be used quite effectively for the transform of all of the nine images, the implementation manages with only three reconfiguration cycles of the FPGA.
As it is not very efficient to calculate the Hough transform within the frequency domain in software, a dedicated software reference has been implemented by *MPS*. The execution of this implementation on the *GR712* took about $942\,\mathrm{s}$ for one single image of 2048×2048 pixels (not including edge-detection).

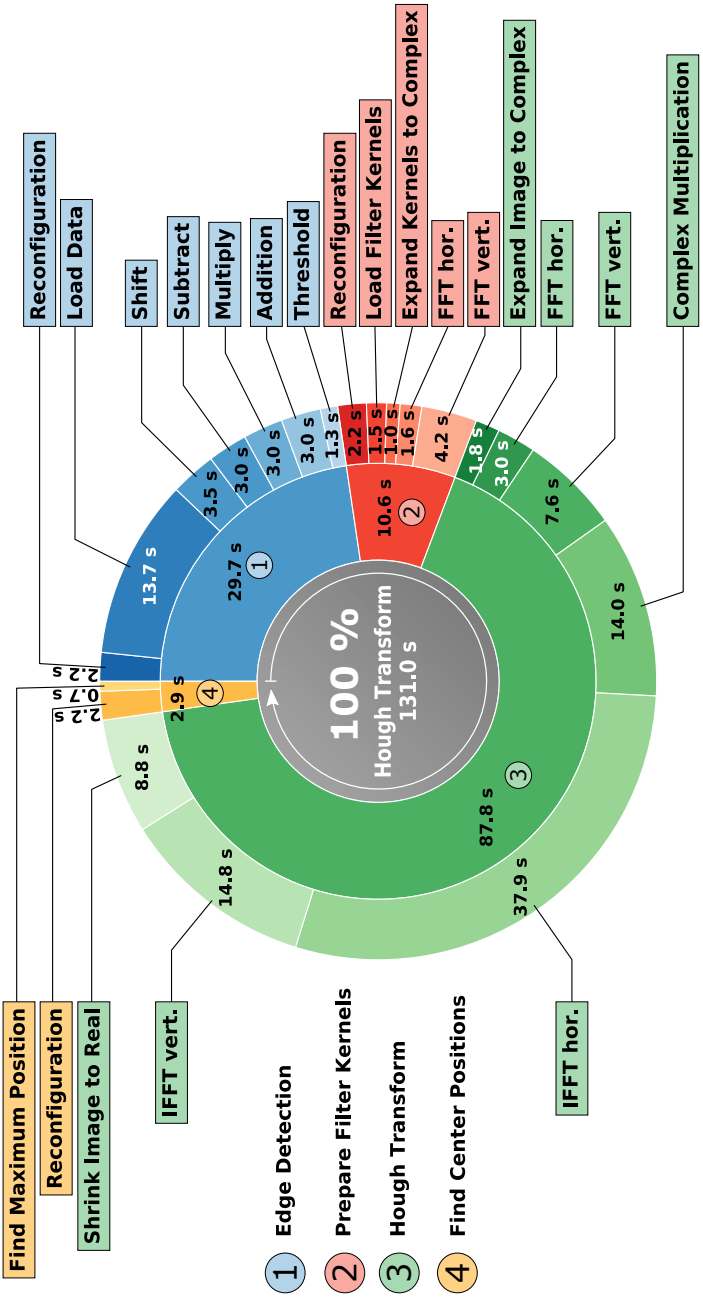


Figure 5.4.: Breakdown analysis of Hough Transform

5.1.5. Kuhn-Lin-Loranz Flat-Field Algorithm

The Kuhn-Lin-Loranz (KLL) iteration which is used for the determination of the flat field is an example for a quite processing intensive algorithm. As described in section 4.0.3, the KLL is implemented using masks, which leads to a high usage of buffer memory. As the flat field is computed iteratively through the dependency of the overall 36 combinations of the nine input images, the calculation contains many loops.
The result of the analysis is depicted in figure 5.5. Due to its principle, the KLL involves many image shifts and multiplications, which is also reflected in the figure. The overall processing time for nine input images is around $500\,\mathrm{s}$. As the final exponentiation of the gain G is carried out in software, a transfer of the image to the *GR712* and the computation of $\exp$ for a single 2048×2048 pixel image is assumed to be around $45\,\mathrm{s}$ (worst case, computation of $\log$ takes around $40\,\mathrm{s}$). As can be seen in the figure, this is negligible compared to the overall run-time of the KLL.

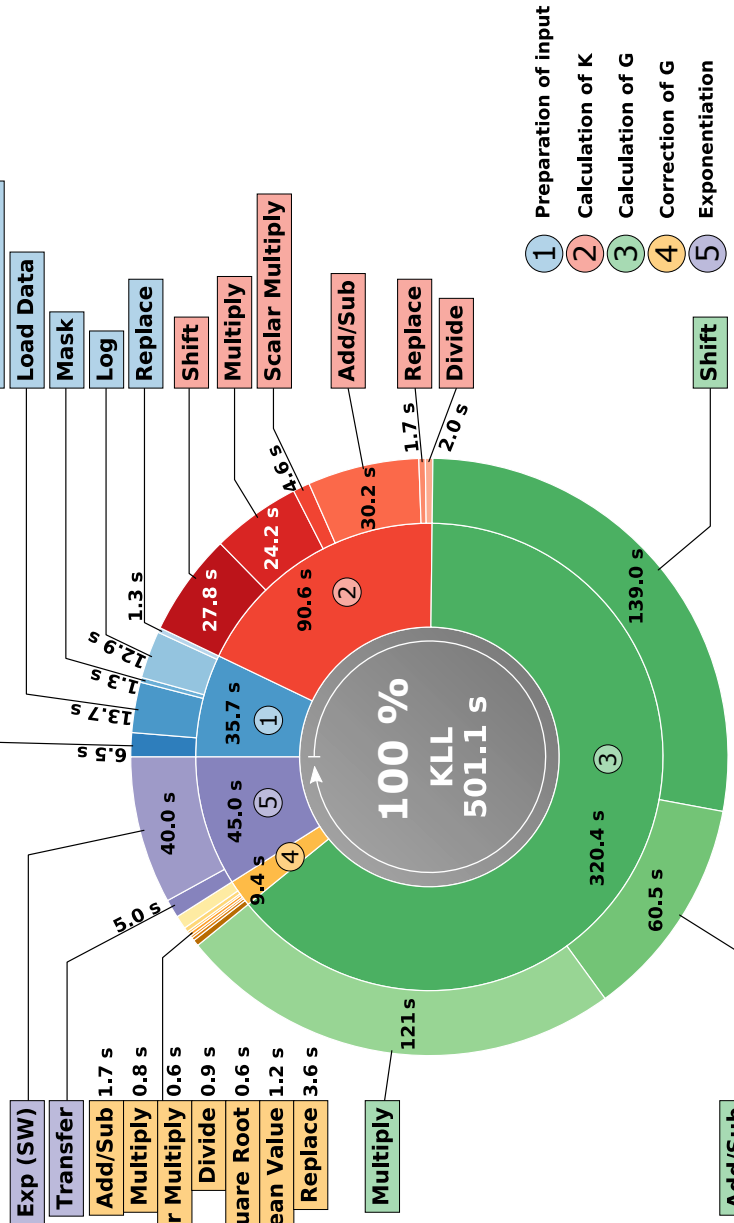


Figure 5.5.: Breakdown analysis of Kuhn-Lin-Loranz Algorithm for 5 iterations

5.1.6. Summary

Summing up, the performance analysis of the presented framework has proven that the execution of the main processing steps, using the *PHI* instrument as an example, can be done in sufficient time. Even though the direct comparison to the existent reference software implementation has to be regarded with care, a significantly increased performance can be observed. It also shows that, instead of using partial dynamic reconfiguration of the used FPGA, a full reconfiguration might be absolutely sufficient. Furthermore, the analysis helps to identify possible bottlenecks which are worth for optimization. It can be observed that most of the implemented functions depend on the performance of the memory interface. Therefore, the overall performance would strongly benefit from higher memory bandwidth, e.g. by increasing the clock frequency.

5.2. Power Estimation

Besides the required processing performance, also the power consumption is typically a limiting factor for spaceborne data processing. To get a general overview of the various devices of the *PHI* DPU and its power consumption, table 5.3 lists the main components and its dissipation in idle state.
The system is divided into two main parts: The base system and the reconfigurable subsystem. While the base system includes the *RTAX2000* FPGA (system supervisor) and the *GR712* CPU (System Controller), the reconfigurable subsystem comprises the two RFPGAs and the associated SDRAM buffer memory. Both include the needed power converters. As a fact, even in an unconfigured state the power dissipation of the reconfigurable system is already quite high.

To measure the consumption on a per-module basis during processing, the current has been monitored during remote execution of processing functions by the *Python* framework. Therefore, the power supply has been read out at central moments of the processing for each function. The average of the power consumption has been calculated for a sample size of $N = 40$ iterations for each module/function. The input supply voltage of the DPU is assumed to be constantly $3.35\,\mathrm{V}$ (remote sensing activated at power supply), the ambient temperature was $25°\mathrm{C}$. The results of this evaluation, the measured current and the resulting power consumption, are shown in table 5.4.
With only a few exceptions, the power consumption of the processing module for each function is around $6.7\,\mathrm{W}$. While a relatively low power consumption can be caused by an underused memory interface (e.g. $\log$ calculation, an increased power consumption might be caused by frequent row activation. This is usually the case for functions operating in the vertical image direction (e.g. vertical FFT, median filtering).

Table 5.3.: Estimated power consumption of sub-components

Component	**Power** [W]
Base System	2.481
Power Converters	0.144
RTAX2000	0.709
GR712RC	0.750
SDRAM Memory (one bank active)	6×0.140
Reconfigurable Subsystem	2.459
Power Converters	0.144
RFPGA1 (unconfigured)	0.567
RFPGA2 (unconfigured)	0.567
SDRAM Memory (idle)	$4 \times 6 \times 0.049$

Table 5.4.: Average current measured at a supply voltage of 3.35 V ($N = 40$)

Module/Function	**Mes. Current** [A]	**Power** [W]
add	2.03	6.70
add_sc	2.02	6.67
sub	2.03	6.70
sub_sc	2.03	6.70
mult	2.03	6.70
mult_sc	2.01	6.63
fp_cmult	2.03	6.70
fp_fft_h	1.85	6.11
fp_fft_v	2.74	9.04
fp_ifft_h	1.85	6.11
fp_ifft_v	2.74	9.04
div	2.03	6.70
div_sc	1.90	6.27
thold	2.01	6.63
real2fpcplx	2.01	6.63
fpcplx2real	2.01	6.63
log	1.56	5.23
sqrt	2.00	6.60
binning	2.31	7.74
fix2float	2.67	8.81
sum	1.97	6.60
lsfit	2.08	6.97
logic	2.03	6.80
logic_sc	2.01	6.73
median_3x3	2.42	7.99
median_5x5	2.15	7.10
crop	2.05	6.77
shift	2.26	7.57
histogram	1.97	6.60

5.3. FPGA Resources

To meet the required timing constraints, a single FPGA configuration can be typically filled up to around 93% for the used *Virtex-4 SX55*. As already mentioned, the complete set of processing modules requires too many resources to fit into a single FPGA design. To overcome this limitation, the presented framework makes extensive use of dynamic reconfiguration. To get an understanding of the overall and per-module resource usage, the generated extended map report files have been evaluated with the help of an automated *Python* script after place and route. The result of this analysis is depicted in figure 5.6.
The diagram shows the percentage of utilization for the essential resources: number of occupied slices, slice registers, Look-Up Tables (LUTs), Block-RAMs and DSP48 blocks. Even though, FIFOs and their associated logic for the read and write ports are located in the memory interface 3.10, the diagram includes these resources in its corresponding modules for better comparison. As can be seen, the limiting factor is the number of occupied slices. Therefore, the diagram is sorted by its usage.
It can be concluded that between three and nine modules could fit into one single configuration bitstream of the used *Virtex-4 SX55*.

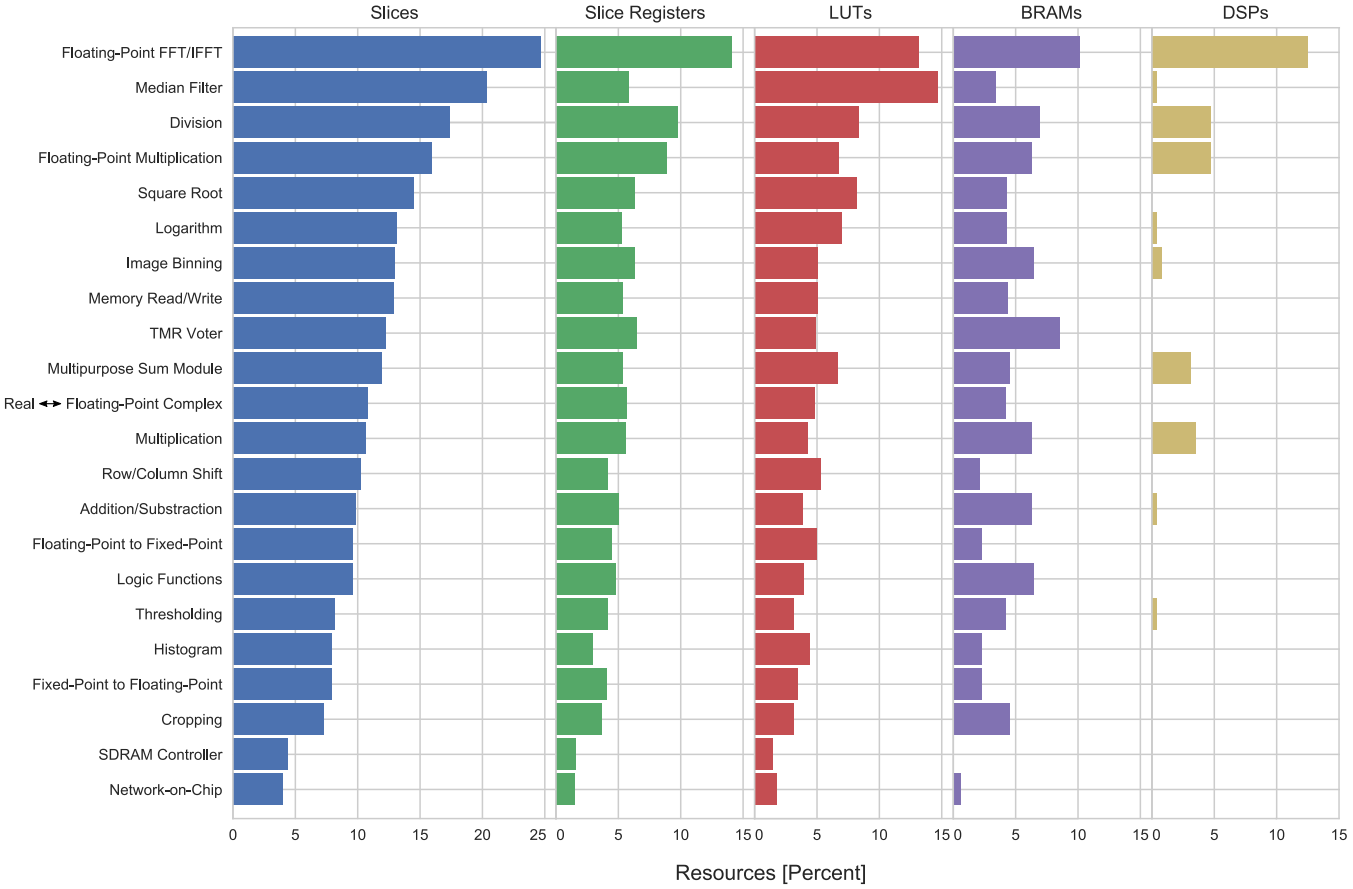


Figure 5.6.: Resource Consumption per Module

6. Summary and Perspective

6.1. Summary

This thesis introduced an approach for a modular, flexible and heterogeneous framework for on-board image data processing. It uses an SRAM-based FPGA with its dynamic reconfigurability at run-time in combination with an attached SDRAM buffer memory. While massive data processing operations are carried out in hardware, the control flow is fully realized in software running on a radiation hardened system controller CPU. The framework furthermore implements an automatic and software-based fault detection which tests each module by means of small test vectors after each execution.
The realization of such an approach has been exemplary presented for the Data Processing Unit of the *Solar Orbiter PHI* instrument. The flexibility and performance has been evaluated based on three examples which represent typical instrument on-board data processing. Among others, this included computation intensive operations, such as matrix multiplication, median filtering, operations in the Fourier domain, the circular Hough transform as well as the determination of the flat field by the Kuhn-Lin-Loranz iteration. It has been proven, that these complex parts of the processing can be subdivided into multiple basic operations, without the need of special customized RTL implementations. Furthermore, the algorithms benefit from the flexible, software-based control flow, which allows the simple adaption the processing to possible changes throughout the mission. It also allows a rapid prototyping and intense evaluation of implemented processing pipelines.
For planning and estimation of the needed FPGA configuration bitstreams, the utilization of FPGA resources has been analyzed on a per-module basis. Furthermore, also the performance of the framework has been evaluated on a per-module basis and compared against the estimation based on theoretical considerations of the underlying hardware implementations. As most of the imple-

mented modules are limited by the performance of the memory interface, this might be a point for further optimization. The run-time has also been compared to a reference software implementation, running on a space-grade *LEON3* CPU. This comparison has shown a significantly improved performance. Separately, the overhead induced by the software-based self-test has been calculated.
Finally, the run-time of the three presented examples has been analyzed in depth. It has been shown that the time spent for full reconfiguration of the FPGAs is negligible compared to the overall run-time of each example. In contrast, transfer of data from and to the locally attached mass memory might play a major role. As a conclusion, all of the three exemplary parts of the processing can be carried out in reasonable time.

6.2. Outlook

Regarding new developments of space-grade processing components, there are only a few candidates which are appropriate for long term deep-space missions like *Solar Orbiter* and will be available in the near future. This includes the *LEON4*-based *Cobham Gaisler GR740* as well as the space-grade *Xilinx Virtex-5* FPGA. As the *GR740* will be available in late 2019, the *Virtex-5QV* is already available. In addition to the *Virtex-5QV*, the *Xilinx Kintex UltraScale* will be available in a space-grade and radiation-tolerant version (*XQRKU060*) soon. First engineering samples are expected to be available by end of 2019. It not only offers even more logic resources compared to its predecessors [94], it also enables the use of *Xilinx*'s High-Level Synthesis (HLS) flow.
Besides classical GPPs and FPGAs, space-grade many-core processors have become available recently. This includes the *Airbus* High-Performance Data Processor (HPDP) [95] as well as the *RC64* by *Ramon Chips* [96]. A direct comparison of these architectures to the presented framework as well as the suitability for a mission like *Solar Orbiter* would be subject of future work.
It has been shown that the presented processing framework is not restricted to the used hardware components exemplary used for the *PHI* instrument. Moreover, it can be used for upcoming missions such as for the proposed *PMI* instrument for the *Lagrange* mission (*ESA*) which is currently under pre-development [97]. The instrument is intended to perform real-time observations of the solar weather. With new components available, the proposed DPM for the instrument could be implemented as shown in figure 6.1. The

architecture would increase the data rate between the processor and the FPGAs by using a PCI interface which would allow memory-mapped access to the buffer memory of the FPGAs as well as DMA (Direct Memory Access) transfer. The performance of the framework would benefit from the use of faster DDR2 SDRAM memory.

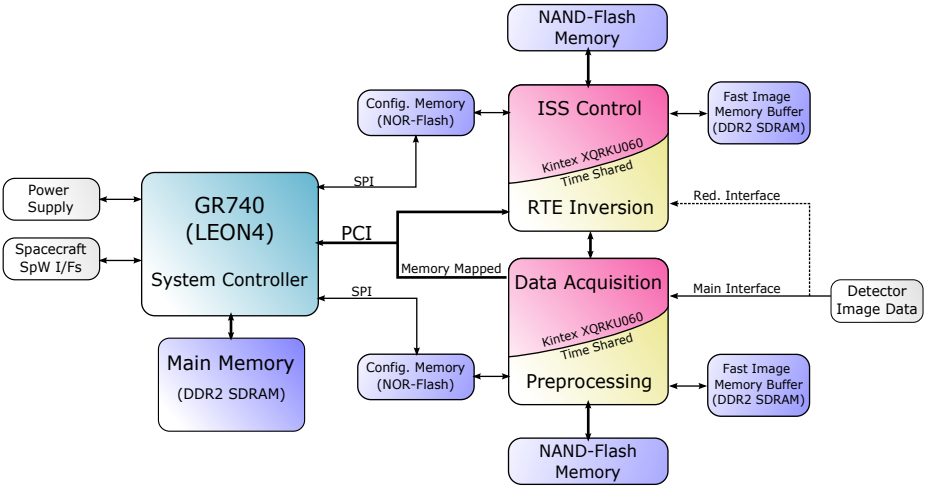


Figure 6.1.: Proposed DPM architecture for *PMI* instrument

A. Runtime and Throughput Estimation

Memory Interface Limited, Standard

The throughput for operations with a linear access to the memory which are limited only by the memory interface can be estimated as follows:

$$\text{throughput}_{\text{MIL}} = \frac{w_{mem}}{n_{pi} \cdot T_{clk}} \cdot (1 - O_{refr} - O_{mport})$$

with $T_{clk} = 20\text{ns}, \quad w_{mem} = 64\text{bit}, \quad l_{burst} = 128, \quad n_{pi} = n_{pw} + n_{pr}$
$l_{cas} = 2, \quad l_{plty,wr} = 3, \quad l_{plty,rd} = l_{plty_wr} + (l_{cas} + 1)$
$O_{refr} = \frac{20}{390}, \quad O_{mport} = \frac{n_p - n_{pi} + l_{plty,wr} \cdot n_{pw} + l_{plty,rd} \cdot n_{pr}}{n_p \cdot l_{burst} + n_p - n_{pi} + 2n_{pw} + 3n_{pr}}$ (A.1)

The equation contains the clock period T_{clk}, net memory bus width w_{mem}, the overall number of memory ports in the FPGA design n_p and the refresh overhead O_{refr}. The refresh overhead consists of 20 clock cycles every 390 cycles. The overhead induced by the multi-port interface is considered by O_{mport}. It incorporates the maximum burst length l_{burst}, the overall number of memory ports in the FPGA configuration n_p, the number of involved ports of the function n_{pi} which consists of the number of write ports n_{pw} and the number of read ports n_{pr}.

Processing Limited

In contrast to the memory interface limited functions, the performance of the processing limited functions are dependent on the implementation of each function. For the applicable functions, logarithm and division, the throughput can be estimated as shown in (A.2).

$$\text{throughput}_{\text{PL}} = \frac{w_{mem}}{n_{cycles} \cdot T_{clk}}, \quad \text{with} \quad n_{cycles,\log} = 30, \quad n_{cycles,div} = 5 \tag{A.2}$$

FFT

For the estimation of the run-time and throughput of the FFT, operation has to be distinguished between horizontal and vertical orientation. The time dedicated to the memory access for reading and writing of data in horizontal mode can be easily estimated using equation (A.1), using one read and one write port. In contrast, the estimation of the vertical mode has to consider the size of a single memory row of the SDRAM and how many rows are activated at the same time (see section 3.4.2 for details). Additionally, the run-time depends on the configured size of the FFT. Table A.1 shows the pure run-time of the FFT as well as the estimated run-time of the memory operations for the horizontal mode for different sizes of the FFT. The number of transform cycles have been determined by simulation. Independent of the FFT-size, the shown run-times correspond to an overall image size of 2048×2048 complex values.

Table A.1.: Run-time FFT vs. memory

Transform Size	Transform Cycles	Estim. FFT Run-Time [s]	Esitm. hor. Memory Run-Time [s]
64	268	0.351	0.197
128	502	0.329	0.197
256	886	0.290	0.197
512	1808	0.296	0.197
1024	3472	0.284	0.197
2048	7340	0.301	0.197

As can be seen in the table, for horizontal operation and small FFT sizes (e.g. 64), the estimated run-time of the FFT core dominates the overall FFT operation. For the run-time of the vertical 2048-point FFT, the run-time seems to be limited by the memory interface. The time for memory operations of the vertical FFT can be estimated to $0.61\,\mathrm{s}$, according to (A.3). Since this estimated value still differs from the observed run-time of $0.82\,\mathrm{s}$, the model is not quite exact for vertical FFT operation.

$$\begin{aligned} \mathrm{T}_{\mathrm{mem,fft}} = &((1 + l_{plty,rd})/2 + l_{plty,wr}) \cdot h_{img} \cdot w_{img} \\ &\cdot T_{clk} \cdot (1 + O_{refr} + O_{mport}) \end{aligned} \tag{A.3}$$

Median Filter

The memory related run-time for the median filtering can be estimated as shown in (A.4). The overhead for multi-port operation and refresh in this case was neglected. The vertical operation of the filter windows has to be considered. The width and height of the input image has to be extended according to the filter size ($w_{img,ext}$ and $h_{img,ext}$). Since the memory access time (table A.2) is faster than the actual median filtering which can be calculated by (A.2), the median filtering is limited by the actual processing.

$$\begin{aligned} \mathrm{T}_{\mathrm{mem,med}} = &\, w_{img,ext} \cdot h_{img,ext} \cdot (f + l_{plty,rd}) \\ &+ w_{img,ext} \cdot h_{img,ext} \cdot (2 + l_{plty,wr}/4)) \\ \text{with} \qquad & \\ w_{img,ext,3\times3} = w_{img} + 2, &\quad h_{img,ext,3\times3} = h_{img} + 2, \quad f_{3\times3} = 1 \\ w_{img,ext,5\times5} = w_{img} + 4, &\quad h_{img,ext,5\times5} = h_{img} + 4, \quad f_{5\times5} = 2 \end{aligned} \tag{A.4}$$

Table A.2.: Run-time Median vs. memory

Filter Size	Cycles per Pixel	Median Run-Time [s]	Memory Run-Time [s]
3×3	6	0.504	0.463
5×5	10	0.842	0.552

Run-time Overview

Table A.3 summarizes the run-time of various processing modules for common image sizes. The measured values include the self-test overhead.

Table A.3.: Measured module run-time for various image dimensions

Module/Function	64	128	256	512	1024	2048
op_addsub						
add()	0.030	0.030	0.030	0.038	0.063	0.168
add_scalar()	0.030	0.030	0.030	0.035	0.053	0.120
sub()	0.030	0.030	0.030	0.038	0.063	0.168
sub_scalar()	0.030	0.030	0.030	0.035	0.053	0.120
op_mult						
mult()	0.030	0.030	0.030	0.038	0.063	0.168
mult_scalar()	0.030	0.030	0.033	0.038	0.053	0.123
op_div						
div()	0.030	0.030	0.030	0.038	0.065	0.170
div_scalar()	0.030	0.030	0.033	0.038	0.060	0.155
op_thold						
thold()	0.053	0.053	0.055	0.058	0.075	0.145
op_cext						
real2fpcplx()	0.028	0.028	0.030	0.038	0.070	0.203
fpcplx2real()	0.028	0.028	0.030	0.038	0.068	0.195
op_fp_fft						
fp_fft_h()	0.028	0.028	0.030	0.045	0.098	0.328
fp_fft_v()	0.028	0.030	0.038	0.068	0.203	0.843
fp_ifft_h()	0.028	0.028	0.030	0.045	0.098	0.328
fp_ifft_v()	0.028	0.030	0.038	0.068	0.203	0.843
op_sqrt						
sqrt()	0.025	0.025	0.025	0.030	0.048	0.118
op_log						
log()	0.028	0.030	0.048	0.113	0.370	1.410
op_fix2float						
fix2float()	0.028	0.028	0.030	0.040	0.073	0.210
op_crop						
crop()	0.030	0.030	0.033	0.040	0.075	0.213

Table A.3 – continued

op_multisum						
lsfit()	0.033	0.033	0.035	0.038	0.055	0.125
sum()	0.033	0.033	0.033	0.035	0.045	0.083
op_logic						
logic()	0.030	0.030	0.030	0.038	0.063	0.168
logic_scalar()	0.028	0.030	0.030	0.035	0.053	0.120
op_shift						
shift()	0.010	0.010	0.013	0.020	0.055	0.193
op_median						
median_3x3()	0.025	0.028	0.030	0.045	0.123	0.515
median_5x5()	0.028	0.030	0.040	0.078	0.235	0.865
op_histogram						
hist()	0.005	0.005	0.005	0.008	0.018	0.053
op_binning						
binning()	0.025	0.025	0.025	0.028	0.038	0.080
fp_cmult()						
fp_cmult()	0.030	0.030	0.033	0.045	0.098	0.310
fp_cmult_scalar()	0.030	0.030	0.033	0.043	0.075	0.213

B. List of Acronyms

Notation	Description
AHB	Advanced High-performance Bus
ALU	Arithmetic Logic Unit
AMBA	Advanced Microcontroller Bus Architecture
AMHD	Analog, Mechanism and Heater Drivers
API	Application Programming Interface
APS	Active Pixel Sensor
ASIC	Application Specific Integrated Circuit
ASMBL™	Advanced Silicon Modular Block
AU	Astronomical Unit
BRAM	Block-RAM
CAN	Controller Area Network
CC	Configuration Controller
CCGA	Ceramic Column-Grid Array
CCSDS	Consultative Committee for Space Data Systems
CLB	Configurable Logic Blocks
CMOS	Complementary Metal-Oxide-Semiconductor
CORDIC	Coordinate Rotation Digital Computer
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CTC	Correlation Tracker Camera
DDR	Double Data Rate
DFPGA	Dynamically Reconfigurable FPGA
DMA	Direct Memory Access
DMIPS	Dhrystone MIPS
DPM	Data Processing Module
DPU	Data Processing Unit
DRAM	Dynamic Random Access Memory
DRPM	Dynamically Reconfigurable Processing Module
DSP	Digital Signal Processing

Notation	Description
DUT	Design Under Test
ECC	Error-Correcting Code
EDAC	Error Detection And Correction
EEPROM	Electrically-Erasable Programmable Read-Only Memory
EGSE	Electrical Ground-Support Equipment
ELE	Electronics Unit
EOP	End of Packet
ESA	*European Space Agency*
FAT	File Allocation Table
FDT	Full-Disc Telescope
FFT	Fast Fourier Transform
FIFO	First In - First Out
FOBP	Fraunhofer On-Board Processor
FOV	Field-of-View
FPA	Focal Plane Assembly
FPGA	Field-Programmable Gate Array
FPU	Floating-Point Unit
FSM	Finite-State Machine
GCR	Galactic Cosmic Ray
GPIO	General-Purpose Input/Output
GPL	GNU General Public License
GPP	General-Purpose Processor
GPU	Graphic Processing Unit
HDL	Hardware Description Language
HK	Housekeeping
HLS	High-Level Synthesis
HPDP	High-Performance Data Processor
HRT	High-Resolution Telescope
HVPS	High-Voltage Power Supply
I^2C	Inter-Integrated Circuit
IAA	*Instituto de Astrofísica de Andalucía*
ICAP	Internal Configuration Access Port
ID	Identifier
IDA	*Institut für Datentechnik und Kommunikationsnetze*
IDL	*Interactive Data Language*

Notation	Description
IEEE	Institute of Electrical and Electronics Engineers
INS	Inertial Navigation System
IO	Input/Output
IP	Intellectual Property
ISA	Instruction Set Architecture
ISS	Image Stabilization System
ITAR	International Traffic in Arms Regulations
JTAG	Joint Test Action Group
KLL	Kuhn-Lin-Loranz Algorithm
LUT	Look-Up Table
MAC	Media Access Control
MAC	Multyply-Accumulate
MBU	Multiple Bit Upset
MCU	Multiple Cell Upset
MIPS	Million Instructions Per Second
MPS	*Max-Planck Institute for Solar System Research*
MTF	Modulation Transfer Function
NaN	Not a Number
NASA	*National Aeronautics and Space Administration*
NoC	Network-on-Chip
NVRAM	Non-Volatile RAM
OBC	On-Board Computer
OCL	On-Board Command Language
OPT	Optics Unit
OTF	Optical Transfer Function
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
PCIe	Peripheral Component Interconnect Express
PCM	Power Converter Module
PDHU	Payload Data Handling Unit
PE	Processing Element
PHI	*Polarimetric and Helioseismic Imager*

Notation	Description
PMP	Polarization Modulator
PROM	Programmable Read-Only Memory
PSF	Point Spread Function
RAM	Random Access Memory
RFPGA	Reconfigurable FPGA
RMAP	Remote Memory Access Protocol
ROM	Read-Only Memory
RTE	Radiative Transfer Equation
RTEMS	*Real-Time Executive for Multiprocessor Systems*
RTL	Register Transfer Logic
RTU	Remote Terminal Unit
S/C	Spacecraft
SDRAM	Synchronous Dynamic Random Access Memory
SECDED	Single-Error Correction and Double-Error Detection
SEE	Single-Event Effect
SEFI	Single-Event Functional Interrupt
SEL	Single-Event Latch-Up
SERDES	Serialization/Deserialization
SET	Single-Event Transient
SEU	Single-Event Upset
SGDR	Safe Guard Data Recorder
SIMD	Single Instruction, Multiple Data
SLC	Single-Level Cell
SMP	Symmetric Multi-Processor
SNR	Signal-to-Noise Ratio
SoC	System-on-Chip
SOI	Silicon on Insulator
SPARC	Scalable Processor ARChitecture
SPI	Serial Peripheral Interface
SRAM	Synchronous Random Access Memory
SSD	Solid-State Drvie
SSMM	Solid-State Mass Memory
SVD	Singular Vector Decomposition
SVN	*Subversion*
SWAP	Size, Weight and Power
TC	Telecommand
TID	Total Ionizing Dose

Notation	**Description**
TM	Telemetry
TM/TC	Telemetry and Telecommand
TMR	Triple Modular Redundancy
TSP	Time-Space Partitioning
TTC	Tip-Tilt Controller
UART	Universal Asynchronous Receiver Transmitter
USB	Universal Serial Bus
VHDL	Very large scale Hardware Description Language
VMC	*Venus Monitoring Camera*
WG	Word Group

C. Bibliography

Author's Publications

[38] T. Lange, F. Bubenhagen, B. Fiethe, H. Michel, and H. Michalik. "FPGA-based Dynamically Reconfigurable Processing Module for the Solar Orbiter PHI Instrument". In: *SEE / MAPLD 2011 Conference*. Apr. 2013.

[39] T. Lange, H. Michel, B. Fiethe, and H. Michalik. "Solar Orbiter Will Process Data Onboard Using Xilinx FPGAs". In: *Xilinx Xcell Journal, Issue 90* (2015). URL: https://www.xilinx.com/publications/archives/xcell/Xcell90.pdf.

[40] T. Lange, H. Michel, B. Fiethe, D. Walter, and H. Michalik. "Fault-tolerant NAND-flash memory module for next-generation scientific instruments". In: *Proc.SPIE*. Vol. 9646. 2015, pp. 9646-9646–7. DOI: 10.1117/12.2195018.

[71] T. Lange, B. Fiethe, H. Michel, H. Michalik, K. Albert, and J. Hirzberger. "On-board processing using reconfigurable hardware on the solar orbiter PHI instrument". In: *2017 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. July 2017, pp. 186–191. DOI: 10.1109/AHS.2017.8046377.

[93] T. Lange, B. Fiethe, Y. Guan, H. Michalik, K. Albert, J. Hirzberger, D. O. Suárez, and M. Rodríguez-Valido. "A flexible and heterogeneous framework for scientific image data processing on-board the Solar Orbiter PHI instrument". In: *Proc.SPIE (in press)*. 2019.

[98] T. Lange, N. Harb, H. Liu, S. Niar, and R. B. Atitallah. "An Improved Automotive Multiple Target Tracking System Design". In: *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*. Sept. 2010, pp. 255–258. DOI: 10.1109/DSD.2010.54.

Author's Contribution

[3] S. K. Solanki et al. "The Polarimetric and Helioseismic Imager on Solar Orbiter". In: *arXiv e-prints*, arXiv:1903.11061 (Mar. 2019), arXiv:1903.11061. arXiv: 1903.11061 [astro-ph.IM].

[10] B. Fiethe, T. Lange, L. Li, H. Michalik, H. Michel, R. Jaumann, A. Lichopoj, N. Schmitz, P. Palumbo, M. Zusi, V. D. Corte, and R. Vitulli. "H/W implementation options for efficient real-time image data compression on the JUICE JANUS camera". In: 2014.

[35] F. Bubenhagen, B. Fiethe, T. Lange, H. Michalik, and H. Michel. "Reconfigurable Platforms for Data Processing on Scientific Space Instruments". In: *Adaptive Hardware and Systems (AHS), 2013 NASA/ESA*. Torino, Italy, 2013.

[37] B. Fiethe, F. Bubenhagen, T. Lange, H. Michalik, H. Michel, J. Woch, and J. Hirzberger. "Adaptive hardware by dynamic reconfiguration for the Solar Orbiter PHI instrument". In: *Adaptive Hardware and Systems (AHS), 2012 NASA/ESA Conference on*. June 2012, pp. 31–37.

[47] H. Michel, A. Belger, T. Lange, B. Fiethe, and H. Michalik. "Read Back Scrubbing for SRAM FPGAs in a Data Processing Unit for Space Instruments". In: *Adaptive Hardware and Systems (AHS), 2015 NASA/ESA*. Montreal, Canada, June 2015.

[54] H. Michel, A. Belger, B. Fiethe, T. Lange, H. Michalik, and M. Kolleck. "How RMAP improves in-flight update of on-board software via SpaceWire". In: *Proceedings of the 6th International SpaceWire Conference Athens 2014*. Athens, Greece, 2014.

[61] K. Albert, J. Hirzberger, D. Busse, T. Lange, M. Kolleck, B. Fiethe, D. O. Suárez, J. Woch, J. Schou, J. B. Rodriguez, A. Gandorfer, Y. Guan, J. P. C. Carrascosa, D. H. Expósito, J. C. del Toro Iniesta, S. K. Solanki, and H. Michalik. "Autonomous on-board data processing and instrument calibration software for the SO/PHI". In: vol. 10707. 2018, pp. 10707-10707–9. DOI: 10.1117/12.2311718.

References

[1] H. Michalik and B. Fiethe. “Integrated Architectures for High Performance Payload Data Processing”. In: *DGLR Jahrbuch*. DGLR, 2005.

[2] D. Müller, R. G. Marsden, O. C. St. Cyr, H. R. Gilbert, and The Solar Orbiter Team. “Solar Orbiter”. In: *Solar Physics* 285.1 (July 2013), pp. 25–70. ISSN: 1573-093X. DOI: 10.1007/s11207-012-0085-7. URL: https://doi.org/10.1007/s11207-012-0085-7.

[4] S. Fahmy, G. Bagnasco, A. Pacros, and K. Wirth. “Solar orbiter payload suite: A hotbed of innovation”. In: *Proceedings of the International Astronautical Congress, IAC*. Jan. 2013, pp. 1291–1300.

[5] ESA. *ESA Solar Orbiter Website*. Feb. 2018. URL: http://sci.esa.int/solar-orbiter/.

[6] W. Markiewicz, D. Titov, N. Ignatiev, H. Keller, D. Crisp, S. Limaye, R. Jaumann, R. Moissl, N. Thomas, L. Esposito, et al. “Venus monitoring camera for Venus Express”. In: *Planetary and Space Science* 55.12 (2007), pp. 1701–1711.

[7] B. Fiethe. *Venus Monitoring Camera - Flight User Manual*. Jan. 2009. URL: VMC-IDA-MA-SF000-001_1.

[8] B. Fiethe, H. Michalik, C. Dierker, B. Osterloh, and G. Zhou. “Reconfigurable System-on-Chip Data Processing Units for Space Imaging Instruments”. In: *2007 Design, Automation Test in Europe Conference Exhibition*. Apr. 2007, pp. 1–6. DOI: 10.1109/DATE.2007.364420.

[9] H. Sierks, H. Keller, R. Jaumann, H. Michalik, T. Behnke, F. Bubenhagen, I. Büttner, U. Carsenty, U. Christensen, R. Enge, et al. “The Dawn framing camera”. In: *Space science reviews* 163.1 (2011), pp. 263–327.

[11] B. Fiethe, A. Dörflinger, and H. Michalik. “Efficient Image Data Processing for the JANUS Instrument on JUICE”. In: *Eurospace DASIA 2019*. 2019.

[12] G. E. Moore et al. *Cramming more components onto integrated circuits*. 1965.

[13] *GR740 Preliminary Datasheet and User's Manual V2.0*. https://www.gaisler.com/doc/gr740/GR740-UM-DS-2-0.pdf. Cobham Gaisler. July 2018.

[14] *RAD750®radiation-hardened PowerPC microprocessor*. BAE Systems. Nov. 2019. URL: https://www.baesystems.com/en-us/download-en-us/20190103202640/1434555668211.pdf.

[15] R. W. Berger, D. Bayles, R. Brown, S. Doyle, A. Kazemzadeh, K. Knowles, D. Moser, J. Rodgers, B. Saari, D. Stanley, and B. Grant. "The RAD750/sup TM/-a radiation hardened PowerPC/sup TM/ processor for high performance spaceborne applications". In: *2001 IEEE Aerospace Conference Proceedings (Cat. No.01TH8542)*. Vol. 5. Mar. 2001, 2263–2272 vol.5. DOI: 10.1109/AERO.2001.931184.

[16] D. Rea, D. Bayles, P. Kapcio, S. Doyle, and D. Stanley. "PowerPC RAD750 - A Microprocessor for Now and the Future". In: *2005 IEEE Aerospace Conference*. Mar. 2005, pp. 1–5. DOI: 10.1109/AERO.2005.1559534.

[17] K. A. Andrew Waterman. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA v2.2*. https://content.riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf. May 2017.

[18] L. Blasi and F. Vigli. "The first space-qualified Klessydra RISCV microcontroller to be launched on a satellite". In: *RISC-V Workshop 2019, Zurich*. June 2019.

[19] *GR712RC Datasheet V2.4*. https://www.gaisler.com/doc/gr712rc-datasheet.pdf. Cobham Gaisler. June 2018.

[20] *RAD5545™ multi-core system-onchip Power Architecture® processor*. https://www.baesystems.com/en/download-en/20181211163917/1434571328901.pdf. BAE Systems. Dec. 2018.

[21] J. M. Dean Saridakis Richard Berger. *RAD55xx Platform SoC*. http://flightsoftware.jhuapl.edu/files/2016/Day-2/Day-2-13-Saridakis.pdf. BAE Systems, Dec. 2016.

[22] *RTG4 FPGAs Product Brief, PB0051*. https://www.microsemi.com/document-portal/doc_download/134430-pb0051-rtg4-fpgas-product-brief. Microsemi. May 2019.

[23] *RTG4 FPGA Datasheet, DS0131*. https://www.microsemi.com/document-portal/doc_download/135193-ds0131-rtg4-fpga-datasheet. Microsemi. Aug. 2018.

[24] F. de Dinechin and B. Pasca. "Reconfigurable Arithmetic for High-Performance Computing". In: *High-Performance Computing Using FPGAs*. Ed. by W. Vanderbauwhede and K. Benkrid. New York, NY: Springer New York, 2013, pp. 631–663. ISBN: 978-1-4614-1791-0. DOI: 10.1007/978-1-4614-1791-0_21. URL: https://doi.org/10.1007/978-1-4614-1791-0_21.

[25] *Radiation-Hardened, Space-Grade Virtex-5QV Device Overview*. https://www.xilinx.com/support/documentation/data_sheets/ds692_V5QV_Data_Sheet.pdf. Xilinx. Jan. 2018.

[26] *NG-MEDIUM NX1H35S Preliminary Data Sheet V1.5*. NanoXplore. 2018.

[27] F. Bubenhagen. "Analysis and Enhancement of a Fault-Tolerant NoC for SRAM-based FPGAs in Space Applications". in press. PhD thesis. TUBS, 2014.

[28] M. Herrmann. *Radiation Characterization of Highly Integrated DDR3 SDRAM Devices for Spaceborne Mass Storage Applications*. Dr. Hut, 2018. ISBN: 978-3-8439-3723-8.

[29] J. Engel, K. S. Morgan, M. J. Wirthlin, and P. S. Graham. "Predicting on-orbit static single event upset rates in xilinx virtex FPGAs". In: (2006).

[30] H. Michel. "Integration of SRAM-FPGAs for Hardware Acceleration of a Data Processing Module for Space Instruments". PhD thesis. 2017.

[31] B. Pratt, M. Caffrey, P. Graham, K. Morgan, and M. Wirthlin. "Improving FPGA Design Robustness with Partial TMR". In: *2006 IEEE International Reliability Physics Symposium Proceedings*. IEEE, 2006. DOI: 10.1109/relphy.2006.251221.

[32] C. Carmichael. “Triple module redundancy design techniques for Virtex FPGAs”. In: *Xilinx Application Note XAPP197* 1 (2001).

[33] *Commercial Products Overview*. SEAKR Engineering, Inc. Oct. 2018. URL: https://www.seakr.com/catalog/.

[34] A. Hofmann, R. Wansch, R. Glein, and B. Kollmannthaler. “An FPGA based on-board processor platform for space application”. In: *2012 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. June 2012, pp. 17–22. DOI: 10.1109/AHS.2012.6268653.

[36] J. Hirzberger and J. Woch. *PHI Instrument User Manual*. SOL-PHI-MPS-OP3000-MA-1, Issue 2, Rev. 1. 2016.

[41] M. Cassel, M. Stähle, U. Lonsdorfer, F. Gliem, D. Walter, and T. Fichna. “The First European Spaceborne Mass Memory System based on NAND-Flash Technology: The Sentinel-2 MMFU”. In: *ReSpace / MAPLD 2011 Conference*. Aug. 2011.

[42] H. Michel, A. Belger, F. Bubenhagen, B. Fiethe, W. Sullivan, A. Wishart, and J. Ilstad. “The SoCWire Protocol (SoCP): A Flexible and Minimal Protocol for a Network-on-Chip”. In: *Adaptive Hardware and Systems (AHS), 2012 NASA/ESA*. 2012.

[43] K. Grürmann, M. Herrmann, F. Gliem, H. Schmidt, G. Leibeling, H. Kettunen, and V. Ferlet-Cavrois. “Heavy Ion sensivity of 16/32-Gbit NAND-Flash and 4-Gbit DDR3 SDRAM”. In: *Radiation Effects Data Workshop*. 2012.

[44] K. Grürmann. “Radiation Characterization of Highly Integrated NAND-Flash Memory Devices for Spaceborne Mass Storage Applications”. PhD thesis. TUBS, 2015.

[45] *NAND Flash Memory, Datasheet*. Micron. 2010.

[46] *Open NAND Flash Interface Specification, Revision 2.2*. ONFI. Oct. 2009.

[48] B. Osterloh, H. Michalik, B. Fiethe, and K. Kotarowski. “SoCWire: A Network-on-Chip Approach for Reconfigurable System-on-Chip Designs in Space Applications”. In: *2008 NASA/ESA Conference on Adaptive Hardware and Systems*. June 2008, pp. 51–56. DOI: 10.1109/AHS.2008.43.

[49] B. Osterloh, H. Michalik, and B. Fiethe. “SoCWire: A Robust and Fault Tolerant Network-on-Chip Approach for a Dynamic Reconfigurable System-on-Chip in FPGAs”. In: *Architecture of Computing Systems – ARCS 2009*. Ed. by M. Berekovic, C. Müller-Schloer, C. Hochberger, and S. Wong. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 50–59. ISBN: 978-3-642-00454-4.

[50] B. Osterloh. “Reliable dynamic partial hardware reconfiguration in space applications”. PhD thesis. 2014.

[51] S. Parkes. *SpaceWire User’s Guide*. STAR-Dundee Limited. 2012. ISBN: 978-0-9573408-0-0.

[52] H. Michel, F. Bubenhagen, and B. Fiethe. *DPU FPGA SoCWire Protocol (SoCP) User Manual*. SOL-PHI-IDA-SW3000-MA-2, Issue 0, Rev. 4. Jan. 2015.

[53] ISO. *ISO C Standard 1999*. Tech. rep. ISO/IEC 9899:1999. 1999.

[55] *RTEMS C User’s Guide*. July 2011. URL: https://docs.rtems.org/releases/rtemsdocs-4.10.1/share/rtems/pdf/c_user.pdf.

[56] T. Wittrock, K.-U. Reiche, and K. Stöckner. “OSIRIS Command Language - A Software System for Flexible Mission Sequencing of Complex Spaceborne Instruments”. In: On-Board Payload Data Processing Workshop. ESTEC, Sept. 2001.

[57] T. Wittrock, K.-U. Reiche, K. Stöckner, H. Michalik, and F. Gliem. “Flexible Operational Sequencing of Complex Spaceborne Instruments – The Software System OCL”. In: *54th Int. Astronautical Congress* (Sept. 2003).

[58] *The OpenCL Specification*. Khronos OpenCL Working Group, May 2018. URL: https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf.

[59] K. Uchiyama. *Design of FPGA-based computing systems with OpenCL*. Cham: Springer, 2018. ISBN: 978-3-319-68160-3. DOI: 10.1007/978-3-319-68161-0.

[60] *SDAccel Environment User Guide, UG1023*. Xilinx, Inc., July 2018. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1023-sdaccel-user-guide.pdf.

[62] J. J. Piqueras Meseguer. "Design and optimization of a space camera with application to the PHI solar magnetograph". PhD thesis. Techn. Univ. Carolo-Wilhelmina Braunschweig, 2013.

[63] J. Piqueras, K. Heerlein, S. Werner, R. Enge, U. Schühle, J. Woch, T. De Ridder, G. Meynants, B. Wolfs, G. Lepage, et al. "CMOS sensor and camera for the PHI instrument on board Solar Orbiter: evaluation of the radiation tolerance". In: *High Energy, Optical, and Infrared Detectors for Astronomy V*. Vol. 8453. International Society for Optics and Photonics. 2012, p. 845314.

[64] J. Hirzberger. *PHI Data Processing Pipeline*. Tech. rep. SOL-PHI-MPS-SW3200-TN-1. Max Planck Institute for Solar System Research, June 2017.

[65] J. C. Carrascosa, B. A. del Moral, J. R. Mas, M. Balaguer, A. L. Jiménez, and J. del Toro Iniesta. "The RTE inversion on FPGA aboard the solar orbiter PHI instrument". In: *Software and Cyberinfrastructure for Astronomy IV*. Vol. 9913. International Society for Optics and Photonics. 2016, p. 991342.

[66] J. C. del Toro Iniesta and B. Ruiz Cobo. "Inversion of the radiative transfer equation for polarized light". In: *Living Reviews in Solar Physics* 13.1 (Nov. 2016), p. 4. ISSN: 1614-4961. DOI: 10.1007/s41116-016-0005-2. URL: https://doi.org/10.1007/s41116-016-0005-2.

[67] J. C. del Toro Iniesta. *Introduction to spectropolarimetry*. Cambridge university press, 2003.

[68] D. O. Suárez and J. Del Toro Iniesta. "The usefulness of analytic response functions". In: *Astronomy & Astrophysics* 462.3 (2007), pp. 1137–1145.

[69] K. Albert, J. Hizberger, J. Woch, and H. Michalik. "On-board calibration of the PHI instrument on-board the Solar Orbiter: Autonomous flat fielding of the HRT". In: (2016).

[70] J. R. Kuhn, H. Lin, and D. Loranz. "Gain calibrating nonuniform image-array data using only the image data". In: *Publications of the Astronomical Society of the Pacific* 103.668 (1991), p. 1097.

[72] *IS42S86400B, IS42S16320B, IS45S16320B, 64M×8, 32M×16 512Mb Synchronous DRAM, Datasheet*. ISSI. 2011.

[73] *GRLIB IP Library User's Manua*. `https://www.gaisler.com/products/grlib/grlib.pdf`. Cobham. Sept. 2018.

[74] T. Kassens. "Implementierung eines FPGA basierenden LEON3 Prozessorsystems für Bilddatenverarbeitung auf dem Solar Orbiter PHI Instrument, Masterarbeit". Inst. f. Datenverarbeitung u. Kommunikationsnetze, TU Braunschweig, 2017.

[75] J. P. C. Carrascosa, B. A. del Moral, J. L. R. Mas, M. Balaguer, A. C. L. Jiménez, and J. C. del Toro Iniesta. "Scientific computing and fault mitigation on FPGA aboard the Solar Orbiter PHI instrument". In: *2015 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. June 2015, pp. 1–8. DOI: 10.1109/AHS.2015.7231150.

[76] J. W. Cooley and J. W. Tukey. "An algorithm for the machine calculation of complex Fourier series". In: *Mathematics of computation* 19.90 (1965), pp. 297–301.

[77] E. Wold and A. Despain. "Pipeline and Parallel-Pipeline FFT Processors for VLSI Implementations". In: *Computers, IEEE Transactions on* C-33.5 (May 1984), pp. 414–426. ISSN: 0018-9340. DOI: 10.1109/TC.1984.1676458.

[78] *LogiCORE IP Fast Fourier Transform v7.1*. http://www.xilinx.com/support/documentation/ip_documentation/xfft_ds260.pdf. Xilinx, Inc. Apr. 2010.

[79] B. Jähne. *Digital Image Processing*. Springer Berlin Heidelberg, 2002. DOI: 10.1007/978-3-662-04781-1.

[80] T. Cormen, C. Leiserson, R. Rivest, M. I. of Technology, C. Stein, I. Books24x7, M. Press, and M.-H. P. Company. *Introduction To Algorithms*. MIT electrical engineering and computer science series. MIT Press, 2001. ISBN: 9780262032933.

[81] D. E. Knuth. *The art of computer programming: sorting and searching*. Vol. 3. Pearson Education, 1997.

[82] R. Mueller, J. Teubner, and G. Alonso. "Sorting networks on FPGAs". In: *The VLDB Journal—The International Journal on Very Large Data Bases* 21.1 (2012), pp. 1–23.

[83] J. Vasquez. *Sort faster with FPGAs*. Jan. 2016. URL: https://hackaday.com/2016/01/20/a-linear-time-sorting-algorithm-for-fpgas/.

[84] C. Larman and V. R. Basili. "Iterative and incremental developments. a brief history". In: *Computer* 36.6 (June 2003), pp. 47–56. ISSN: 0018-9162. DOI: 10.1109/MC.2003.1204375.

[85] T. Schuster. "SoCRocket - Eine flexible erweiterbare Virtuelle Plattform zum Entwurf robuster Eingebetteter Systeme". PhD thesis. 2015.

[86] J. Ernesti and P. Kaiser. *Python 3 - Das umfassende Handbuch*. Jan. 2017. ISBN: 978-3-8362-5864-7.

[87] K. Albert. *Processing Pipeline Algorithms Descriptions*. Tech. rep. SOL-PHI-MPS-SW4400-TN-1. Max Planck Institute for Solar System Research, Feb. 2019.

[88] P. V. Hough. "Machine analysis of bubble chamber pictures". In: *Conf. Proc.* Vol. 590914. 1959, pp. 554–558.

[89] P. V. Hough. *Method and means for recognizing complex patterns*. US Patent 3,069,654. Dec. 1962.

[90] R. O. Duda and P. E. Hart. “Pattern classification and scene analysis”. In: *A Wiley-Interscience Publication, New York: Wiley, 1973* (1973).

[91] C. Hollitt. “Reduction of computational complexity of Hough transforms using a convolution approach”. In: *Image and Vision Computing New Zealand, 2009. IVCNZ'09. 24th International Conference*. IEEE. 2009, pp. 373–378.

[92] C. Hollitt. “A convolution approach to the circle Hough transform for arbitrary radius”. In: *Machine Vision and Applications* 24.4 (May 2013), pp. 683–694. ISSN: 1432-1769. DOI: 10.1007/s00138-012-0420-x. URL: https://doi.org/10.1007/s00138-012-0420-x.

[94] *UltraScale Architecture and Product Data Sheet: Overview*. https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf. Xilinx. May 2019.

[95] O. Dokianaki, V. Baumgarte, M. Syed, C. Papadas, T. Helfers, G. Dramitinos, T. Scholastique, M. Kogan, I. Saenger, S. Lacan, O. Markakis, and L. Hili. “HPDP: Architecture and design flow: High performance data processor”. In: *2017 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. July 2017, pp. 62–70. DOI: 10.1109/AHS.2017.8046360.

[96] R. Ginosar, P. Aviely, H. Gellis, T. Liran, T. Israeli, R. Nesher, F. Lange, R. Dobkin, H. Meirov, and D. Reznik. “RC64, a Rad-Hard Many-Core High-Performance DSP for Space Applications”. In: *DASIA 2015-DAta Systems in Aerospace*. Vol. 732. 2015.

[97] J. P. L. S. Kraft K. G. Puschmann. “Remote sensing optical instrumentation for enhanced space weather monitoring from the L1 and L5 Lagrange points”. In: vol. 10562. 2017. DOI: 10.1117/12.2296100. URL: https://doi.org/10.1117/12.2296100.